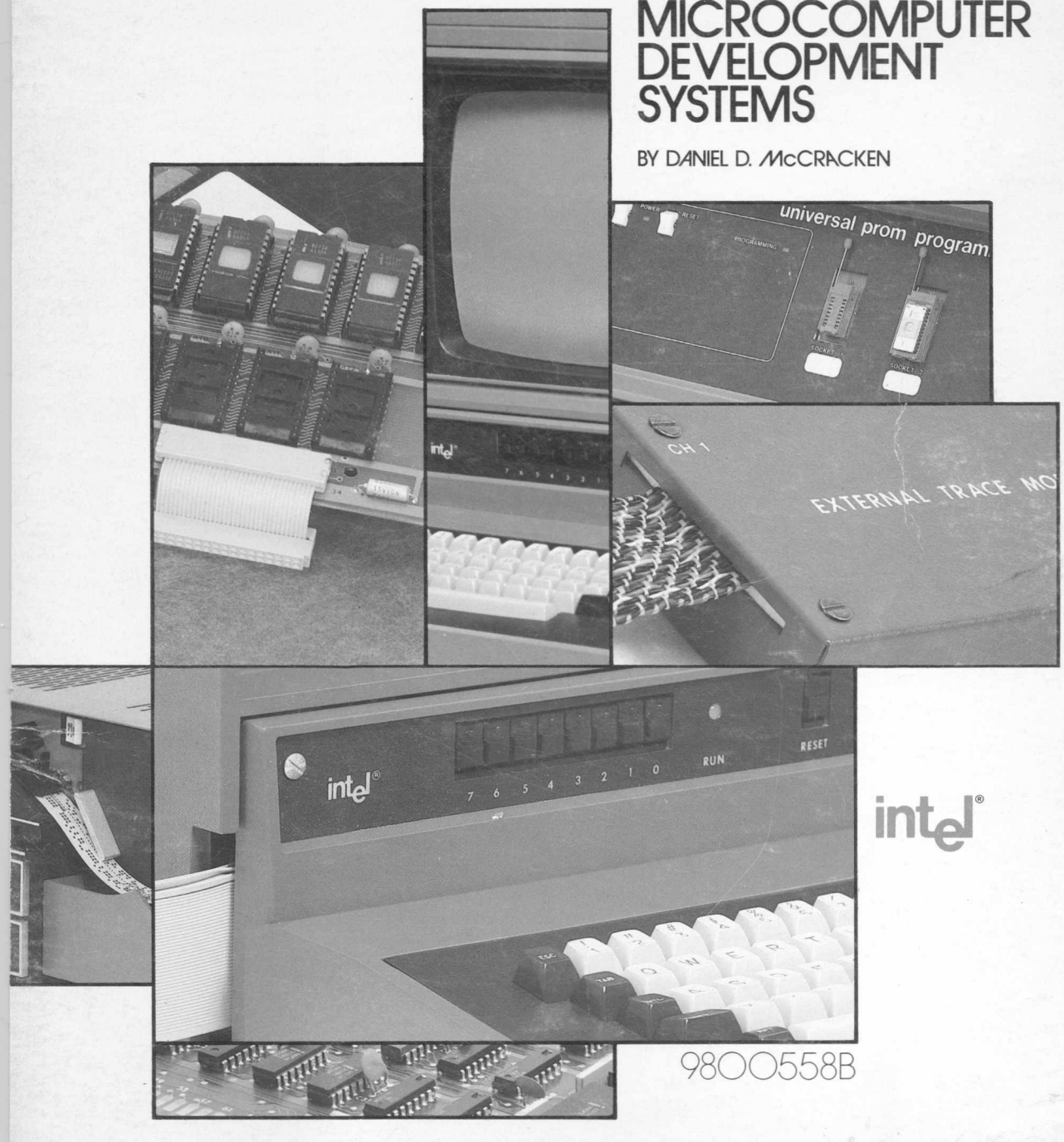
A guide to INTELLEC® MICROCOMPUTER DEVELOPMENT SYSTEMS

BY DANIEL D. McCRACKEN



intel®

9800558B

ABOUT THE AUTHOR

Daniel D. McCracken is a leading author of textbooks on computer programming. His *Digital Computer Programming* (1957) was the first text on the subject. Among his 15 titles are standard works on Fortran (1961, 1965, 1972, and 1974), Algol (1962), COBOL (1963, 1970, and 1976), and numerical methods (1964 and 1972). His latest book is *A Guide to PL/M Programming for Microcomputer Applications* (Addison-Wesley, 1978).

He graduated in 1951 from Central Washington University with degrees in mathematics and chemistry. After seven years with the General Electric Company in a variety of assignments in computer applications programming and programmer training, he spent a year at the New York University Atomic Energy Computing Center, then went into full time consulting and writing on computer subjects. His clients have included Honeywell, The RAND Corporation, Shell Oil, IBM, and Intel.

He writes frequently for Datamation and other leading publications in data processing.

He is vice president of the Association for Computing Machinery, has been chairman of the ACM Committee on Computers and Public Policy, and is a three-time ACM National Lecturer. He occasionally teaches a course at Columbia University. □

A GUIDE TO INTELLEC MICROCOMPUTER DEVELOPMENT SYSTEMS

Chapter 1

AN OVERVIEW OF INTELLEC MICROCOMPUTER DEVELOPMENT SYSTEMS

INTRODUCTION: *A WORD TO THE READER*

This manual is for the person who wants an easy introduction to the Intellec Series II Microcomputer Development System: why it is needed, what it does for you, what its major components are (hardware and software), with simple examples that will help you get a rapid start using it.

We assume that you have some basic knowledge of microprocessors and their applications, but few demands are made on that knowledge. We assume that you have had some minimum exposure to programming, but we do not assume familiarity with any particular language.

The intent of the manual is to give you a clear idea of what an Intellec Microcomputer Development System can do for you, and, if you already have one, get you off to a fast start using it effectively in your own work.

ABOUT THIS MANUAL

After this introductory chapter, which gives you a bird's-eye view of the system, the bulk of the manual is devoted to five chapters giving you brief sketches of the major components.

Chapter 2 covers the monitor, ISIS, and files; Chapter 3 deals with the text editor; Chapter 4 is a brief introduction to PL/M; Chapter 5 is devoted to the MCS-80/MCS-85 macro assembler; Chapter 6 explains the basic concepts of in-circuit emulation. Chapter 7 then walks through the complete development cycle with a small application.

In every case there are numerous examples. In four chapters an illustrative console session is reproduced. In all chapters there is material that you can try yourself, without any other manuals, if you wish. (Of course, before running these examples you must install your system according to the instructions in *Intellec Series II Installation and Service Manual*.) The simple example in the next chapter lets you have something running on your system with a minimum of effort and background.

In other words, this manual will help you get off to a running start. Naturally, you will need the manuals listed in Appendix II also, but this manual contains enough of the basics to let you get your feet on the ground, and serves as a road-map to the other manuals.

Good reading!

THE MICROCOMPUTER DEVELOPMENT PROCESS

Designing a product containing a microcomputer requires close coordination of two separate but highly interdependent design efforts, hardware development and software development. Hardware development involves planning the interaction of the microprocessor chip itself, the associated memory and peripheral circuits, and the specialized input/output circuits and chips. Software development involves programming the microprocessor, using instructions that will eventually be stored in the product's memory, to correctly perform the required tasks.

These two development efforts might — in the abstract — be carried out independently. In practice, it is usual — and highly preferable — for them to be carried out in parallel. Both to save time and to achieve good system integration, software debugging must usually begin long before there is completed prototype hardware on which to test it.

THE FUNCTIONS OF A MICROCOMPUTER DEVELOPMENT SYSTEM

To carry out the various tasks of the microcomputer development cycle effectively and efficiently, it is necessary to have supporting development tools, both hardware and software. There must be a computer system on which to write programs, compile or assemble them, and store them during development. Entering programs and other text files requires a text editor, and there need be facilities for manipulating files in various other ways. There has to be some way to test programs as they are being written, at a time when prototype hardware may be incomplete or not available at all.

The Intellec Series II Microcomputer Development System answers these needs, providing you all the tools you need to bring your microcomputer application to successful operation in as short a time as possible. Key features include:

- The Intellec Series II system provides all the hardware and software support you need to implement products using the Intel MCS-80, MCS-85, and MCS-48 microprocessor families. It will support future Intel microprocessors as they are introduced.

- The Intellec Series II system contains its own microprocessor, memory, high-speed peripherals, a diskette-based operating system, and development software; together they provide everything you need to design and debug your software.

- In-circuit emulation makes it possible to debug your hardware with the help of flexible software debugging aids, and lets you — through sharing of Intellec hardware resources — debug your software while the prototype hardware is still being designed and built.

- The Intellec Series II product line is modular in configuration so that you can tailor your system to your particular needs and budget, then expand it easily when you choose.

IN-CIRCUIT EMULATION

The single feature providing you the greatest ease and versatility in developing your product is an In-Circuit Emulator (ICE) module. ICE modules let you test your software using however much of your hardware — including *none* — that is available at each stage of the development process. Using ICE, you can begin testing your software before any prototype hardware even exists. Then as portions become available you can use them, "borrowing" resources (memory, input/output) from the Intellec system to fill in for prototype hardware not yet ready. When all prototype hardware is running, you can use ICE for a thorough debugging of it.

In-circuit emulation modules are available to support systems based on the 8080, 8085, 8048, and Series 3000 microprocessors. As new CPUs are developed, matching in-circuit emulation packages will be made available. However, due to processor differences, not all of the features in the following general description are available in every ICE system.

With in-circuit emulation, you control, interrogate, revise, and completely debug your product *in its own environment*. When you plug ICE into your system in place of its processor, you gain all the diagnostic power and flexibility built into the Intellec system.

You don't need to build your own special display or debugging hardware to support development, since these are integral parts of the Intellec system. You can use Intellec memory and input/output facilities in the early phases of testing your prototype. Then, on a step-by-step basis, you replace shared resources with prototype-resident memory and I/O.

In-circuit emulation allows you to set hardware and software conditions under which program execution is to be stopped; these are called breakpoints. External breakpoint probes are provided to attach to any logic signal on a prototype board to detect hardware conditions not directly accessible to the microprocessor bus.

In-circuit emulation provides the capability to examine and alter CPU registers, main memory, pin and flag values, and automatically collect and store address, data and status information for machine cycles emulated. You can examine and modify your program using symbolic references instead of absolute values.

As you will see in Chapters 6 and 7, this symbolic debugging is one of the key features of Intellec Microcomputer Development Systems. In essence, it lets you debug your program at the source program level, that is, in terms of the program as you wrote it. There is no need to be constantly converting your symbols to absolute machine addresses.

THE OPERATING SYSTEM

Intel System Implementation Supervisor (ISIS) is a diskette-based package of development software and file handling facilities designed for use with the Intellec system and specifically tailored to the needs of microcomputer development. Key features include:

- The ISIS text editor provides string search, substitution, insertion, and deletion commands.
- The 8080/8085 macro assembler generates relocatable code.
- The LINK program permits object programs, written in any Intel programming language, to be combined, resolving external references in the process.
- The LOCATE program converts a relocatable object program to specific memory locations, with full control over where the program code, stack, and storage areas of a program are placed.
- The Library Manager simplifies use of object program libraries in the development of large programs.

In other words, programs can be created, edited, assembled, debugged and executed — all without paper tape handling. Program listings can be directed to diskette for later use, or printed on the line printer.

ISIS is standard with all Intellec Series II Model 220 and 230 Systems. For Model 210 users a ROM-based monitor/editor/assembler provides all necessary system functions.

THE PL/M LANGUAGE AND COMPILER

PL/M is a high-level language that is particularly well suited for use in system programming. It has a number of advantages over programming in the language of the microprocessor itself. Programs written in PL/M are generally faster and less expensive to develop, more reliable, easier to understand and check out, and simpler to maintain. A modern macro assembler is provided for those who choose to program in assembly language, but increasing numbers of users of Intel MCS-80 and MCS-85 microprocessors are turning to PL/M to gain these advantages.

The PL/M compiler runs directly on the Intellec system using the ISIS operating system, producing efficient relocatable code which can be easily linked with other PL/M and/or assembly language programs.

With PL/M you can create, compile, modify, link, relocate and debug programs entirely on the Intellec system itself with no requirements for large in-house computers or time-sharing services.

THE FORTRAN-80 LANGUAGE AND COMPILER

Fortran is a high-level language that is particularly well suited to application programs. Intel's Fortran-80 compiler implements the ANSI Fortran-77 standard. To the various advantages of PL/M-80 just cited, it adds powerful arithmetic processing capability and a variety of facilities for handling formatted input and output.

Fortran-80 can be used both as a development tool, while programs are being tested and run in the Intellec environment, and for application programming. In either case, one is free to develop modules in whatever language (8080/8085 Macro Assembler, PL/M-80, or Fortran-80) has the greatest strengths for a given portion of the task; ISIS provides the facilities for linking together the object programs produced by the assembler and compilers.

IN SUMMARY

Using the Intellec Series II Microcomputer Development System, hardware and software development converge as early in the development cycle as possible. The various software tools (the text editor, the MCS-80, MCS-85, or MCS-48 macro assemblers or the PL/M or Fortran compilers, the file management modules) allow you to develop your software in a quick, convenient manner. Hardware tools, such as in-circuit emulation, allow you to debug your software and hardware together.

Your configure your Intellec system to meet *your* needs. The Intellec product line ranges from a simple ROM-based system to a highly sophisticated combination of advanced hardware and software. You can begin with whatever level meets your present needs, then easily migrate to more complete versions as your changing needs require.

Chapter 2
THE MONITOR AND ISIS

The hardware of the Intellec system can't do anything by itself — it needs software to tell it what to do. We are speaking now of the software provided by Intel as part of the total package, not the programs you write for your application. In order for the Intellec system to help you develop your programs, there have to be other programs already in the system, so to speak, to make it run.

The system software comes in two parts, the *monitor* and *ISIS* (Intel System Implementation Supervisor).

THE MONITOR

The monitor is a program that controls the central processing unit (CPU) of the Intellec system at the most elementary level. The monitor program is supplied as a ROM chip set; it is permanently installed in the Intellec chassis, ready to go to work when you turn on the equipment and carry out the simple operations to get things started. It carries out functions such as these:

- Control of input and output. The monitor provides software to interface with all standard Intellec peripherals.
- Displaying and/or modifying the contents of Intellec memory.
- Displaying and/or modifying the contents of internal CPU registers.
- Loading other software that is described below.
- Executing any program in Intellec memory, either RAM or ROM.

Telling the monitor what we want it to do is a matter of typing simple commands at the system console, once the system has been initialized.

For example, reading a paper tape in hexadecimal format from the paper tape reader is as simple as typing the command,

```
R0
```

where R is the coded command for READ, and the zero has a function that is not important to us.

Another example of a monitor function is the Display command. Suppose we wish to see the contents of memory locations 2000 through 2010, both numbers being hexadecimal. We enter the command,

```
D2000,2010
```

There are various other commands for carrying out the functions sketched at the beginning of this chapter.

AN ILLUSTRATIVE MONITOR SESSION

The best way to get a rapid idea of what the monitor does, is to see it in action. To that end, we present the following sample terminal session, in which a small program is entered into Intellec memory and executed. You are encouraged to enter the program yourself; with reasonable care and a bit of luck with your typing, you can have your system doing something visible under your control, within 20 minutes!

The program that we shall enter is in fact the program shown in Fig. 5-1, which sends the letter X to the system console once each second until interrupted from the Intellec front panel. The program as shown there is *relocatable*, which means that it can be set up to execute from any part of Intellec memory, using the LOCATE command that will be discussed later. We shall work with the program as located to run from memory locations 4000H-4021H.

The dialog of the session is shown on the facing page, with explanatory comments keyed to bold face numbers in the margin of the terminal printout.

1. I initialize the system, which requires only turning on the Intellec components. (In the Model 220 and Model 230 the disk drive doors must be open to run this example.) The monitor responds with an identifying message and a "dot prompt," meaning that it prints a dot (period or decimal point, if you prefer) to announce that it is ready to accept commands.

2. I will want to enter the program using the Substitute command, which is a bit less confusing if the previous contents of the memory locations being changed are all zero. Accordingly, I specify that memory locations 4000 through 4FFF (both numbers always hexadecimal) be filled with zeros, using the Fill command. There was no real need to fill so much memory, but the operation is so fast one doesn't worry about the time wasted.

3. Using the Substitute command four times I enter the program, one byte at a time. After the letter S (for Substitute) I give the starting location of the program, 4000. When I press the space bar the monitor prints the current contents of location 4000, which is zero because of the prior Fill operation; I type 31, which is the hex representation of the byte I want substituted for the zero. When I press the space bar again, the monitor prints the contents of location 4001, which I replace with 30. Proceeding in this way I enter the first ten bytes of the program, then press carriage return and start again with 400A, the next hex location after the one at the end of the previous line. All 34 bytes of the program are entered in the same way.

4. I tell the monitor to begin executing instructions at 4000 (always hexadecimal), which is the start of the program I just entered. It does so, sending the letter X to the system console once each second.

5. To stop the program, I press the Interrupt 0 key on the Intellec front panel. When this is done, the instruction then being executed is completed, the CPU registers are stored, and control of the system is given to the monitor. The hex address after the crosshatch (#) is that of the next instruction to be executed. (If you run the program, you may get a different address because the program is in a different place when you interrupt it.)

6. Using the X command, I ask for the contents of all CPU registers. The letters A through E are the registers with those names; F is the program flag byte; H and L are the memory address registers; I is the 8080 interrupt mask; M is the combination of the H and L registers; P is the program counter, giving the location of the instruction to be executed next; S is the stack pointer. I observe that the program counter is the same as that given when I interrupted the program.

7. I say Go, without specifying an instruction address. Program execution is continued from where it left off. X's are again sent to the CRT.

8. I again interrupt the program — sooner this time — which stops at the same location as before. (As it happens, the program spends most of its time in a small loop at this location.)

9. I again ask for the register contents, some of which are now different because of the accident of when the program was interrupted.

10. Using the Substitute command, I change the contents of a memory location. The effect will be to ring the bell in the system console instead of printing an X.

```
1  MDS MONITOR, V2.0

2  .F4000,4FFF,00
3  .S4000 00-31 00-30 00-40 00-16 00-32 00-3E 00-C8 00-CD 00-16 00-40
   .S400A 00-15 00-C2 00-05 00-40 00-0E 00-58 00-CD 00-09 00-F8 00-C3
   .S4014 00-03 00-40 00-06 00-0C 00-48 00-0D 00-C2 00-19 00-40 00-3D
   .S401E 00-C2 00-18 00-40 00-C9
4  .G4000
5  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX#4019
6  .X
    A=A3 B=0C C=04 D=09 E=EE F=12 H=12 I=FE L=34 M=1234 P=4019 S=402E
7  .G
8  XXXXXXXX#4019
9  .X
    A=35 B=0C C=04 D=1C E=EE F=12 H=12 I=FE L=34 M=1234 P=4019 S=402E
10 .S400F 58-07
```

11. The program is started from where it was interrupted. The bell dutifully rings once each second.

12. An interrupt again catches the program at the same instruction as before.

13. I use the Display command to get a copy of the entire program, 16 numbers to a line.

14. Using the Assign command I specify that the listing device should be the line printer on my system, rather than the console device which is the default.

15. Now, when I ask for the program to be displayed, it is printed on the line printer.

16. I substitute a different byte for the one at 4004.

17. I use the Go command with a breakpoint. The command says to start at 4000, but to stop just before executing the command at 4007.

18. The program stops as ordered.

19. I want to see if the registers have the values I expect, knowing what I want the first three instructions to do. Everything is in order.

20. I say Go, without a breakpoint; execution begins with the instruction at 4007, where we stopped. The effect of the last change ws to slow the program down so it rings the bell once each *five* seconds.

21. I interrupt the program, which this time stops at a different instruction.

22. With nothing else to do, I turn off power on the Intellec components.

THE ISIS SYSTEM

The Intel System Implementation Supervisor (ISIS) is a collection of programs that facilitate the development of microcomputer software. It is supplied on diskette. Among the functions of ISIS are the following, most of which are discussed later in this manual and more fully in other Intel publications.

- All of the functions of the monitor are still available.

- Commands for moving, renaming, deleting, and listing names of *files*, a file being any named program or collection of data. Examples of ISIS file commands follow at the end of this chapter.

- A complete system called the *text editor*, which is used to create files — such as programs — as well as to modify them in a variety of ways.

- The 8080/8085 Macro Assembler translates programs written in assembly language to the machine language of the Intel MCS-80/MCS-85 family. The assembler is itself a program, which is brought into the Intellec memory from diskette when you type the command ASM80 and give the name of the file containing the assembly language program to be translated to machine language.

- Two programs named LINK and LOCATE that greatly facilitate program development, making it possible to combine programs and to prepare them for execution from any memory location. Their functions are discused in connection with the assembler and PL/M, in Chapters 4, 5, and 7.

- The Library Manager, which makes it possible to place the object code for frequently used programs into a library, then specify the name of the library for the LINK operation instead of having to name all the individual programs.

```
11 .G
12 #4019
13 .D4000,4021
   4000 31 30 40 16 32 3E C8 CD 16 40 15 C2 05 40 0E 07
   4010 CD 09 F8 C3 03 40 06 0C 48 0D C2 19 40 3D C2 18
   4020 40 C9
14 .AL=L
15 .D4000,4021
16 .S4004 32-FF
17 .G4000 -4007
18 #4007
19 .X
    A=C8 B=0C C=01 D=FF E=EE F=12 H=12 I=FE L=34 M=1234 P=4007 S=4030
20 .G
21 #401A
22 .
```

ISIS OPERATIONS WITH FILES

In all of your work with the Intellec system, you will be dealing with files and using certain ISIS commands heavily. We accordingly present further information about files and the ISIS commands for handling them.

An ISIS file is any named collection of bytes. A file is usually stored on diskette in an Intellec system, but it can also be a set of data coming from a peripheral device. Examples of files:

- A program that you enter using the text editor (described in the next chapter).
- A set of sample data, also prepared with the text editor.
- The output of a program, written to diskette.
- The contents of a paper tape, whether a program or data, that you borrowed with the permission of someone else.

Usually, a file is a program, in any of several forms, as we shall discuss later.

You will most commonly create a file using the text editor. One of the ISIS commands is EDIT, which must state the name we want the file to have. The complete name consists of three parts; a *device identifier* enclosed in colons, the *name*, and an optional *extension* separated from the name by a period.

Example of file names:

```
:F1:2FURN.SRC

:F1:2FURN.OBJ

:F1:ONESEC

:F2:ANDER6.LST
```

EXAMPLES OF ISIS FILE COMMANDS

The COPY command permits you to move a file from one place in the Intellec system to another. Perhaps you want to print a file on your line printer, using this command:

```
COPY :F1:ONESEC.LST TO :LP:
```

Or perhaps you wish to have a second copy of the file, with a different extension, on diskette:

```
COPY :F1:ONESEC.SRC TO :F1:ONESEC.SAV
```

The RENAME command lets you change the name of a file on diskette, as in this example:

```
RENAME:F1:ONESEC.SRC TO :F1:1SEC2.SRC
```

The DELETE command lets you remove files from diskette, making the storage space available for other files, as in this example:

```
DELETE :F1:ONESEC.SRC
```

The DIR command displays a directory of all of the files on a diskette, as in this example, which refers to diskette 1:

```
DIR 1
```

The LINK command combines files of object programs into a new — and also relocatable — object program. For an example, suppose you have a program named :F1:MAIN.OBJ that refers to a subprogram named :F1:SUB.OBJ and which needs to borrow ISIS procedures that are in a library named SYSTEM.LIB. You want to name the combined file :F1:PROG.OBJ. The ISIS command:

```
LINK :F1:MAIN.OBJ, :F1:SUB.OBJ, SYSTEM.LIB TO :F1:PROG.OBJ
```

As you can see, the LINK facility makes it possible to combine programs — possibly written by different programmers — with procedures in libraries (either ISIS or those you create yourself using the Library Manager). This can be helpful even in writing small programs, and is essential in a large project with many programmers.

The LOCATE Program converts a relocatable object program into absolute form, that is, it assigns all of the parts of the program to fixed memory locations. If we don't care where the program is placed, as will often be the case during program development, the command can be as simple as

```
LOCATE :F1:PROG.OBJ
```

The output is a new file having no extension: :F1:PROG. On the other hand, an application program going into hardware having several different types of memory might use some of the flexibility of the command, as in this example:

```
LOCATE :F1:PROG.OBJ CODE(4000H) DATA(6000H) STACK(6200H)
```

Even more flexibility is available.

SUMMARY

The monitor, which is permanently available in ROM, provides basic capabilities for using your system. ISIS, which can be used on any Intellec system having disk drives, greatly extends the range and power of system software.

Chapter 3

THE TEXT EDITOR

The Intellec text editor lets you create and edit files of text. Examples of text files are source programs written in assembly language or PL/M, tables to be used in a computation, or project documentation. You will probably use the text editor most commonly to prepare and correct programs.

Before we delve into some of the details, here is an overview of how the text editor is used. After the equipment has been turned on you initiate execution of the editor. You type in your program, making changes and corrections as you go; when you are finished creating the file you store it on diskette (or punch it onto paper tape in the Model 210). The text file can now be used however you wish: you can assemble or compile it, if it is a program, or you can ask for a listing on the console or line printer if you have one, or you can call on the text editor again and make changes in the file.

THE POINTER

Once you have entered some text, all further operations depend on a *pointer* to the text. Since the text editor is character-oriented, the pointer locates a character that is to be acted upon. The pointer may be positioned before the first character of the text, between two characters, or after the last character. The pointer is never positioned directly *at* a particular character, but always *between* two characters. When text is entered it is placed at a point immediately following the pointer. As each character is entered, the pointer is moved to a position immediately following that character.

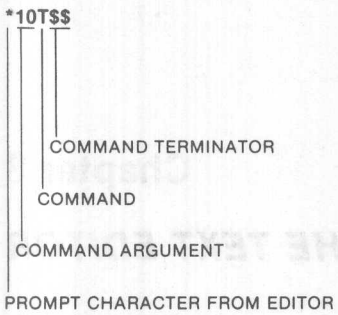
You can move the pointer to any position in your text, using commands that are illustrated below. Pointer movement may be in terms of characters or complete lines. Considering text in terms of lines is convenient because most text is divided into lines.

COMMANDS AND COMMAND STRINGS

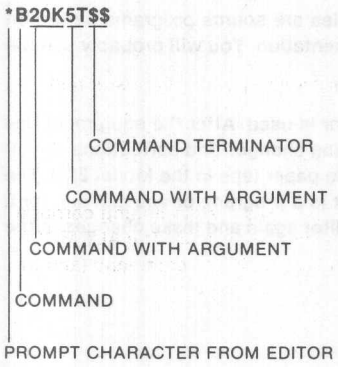
Each editor *command* consists of a single letter. Certain commands take *arguments*. Commands may be entered one at a time or may be combined into *command strings*. The text editor signals its readiness to accept commands by printing a prompting asterisk in the leftmost column of the system console device. Command strings must be terminated with a pair of ALT MODE (alternate mode) or ESC (escape) characters (depending on the type of the console device.)

EXAMPLES OF COMMANDS

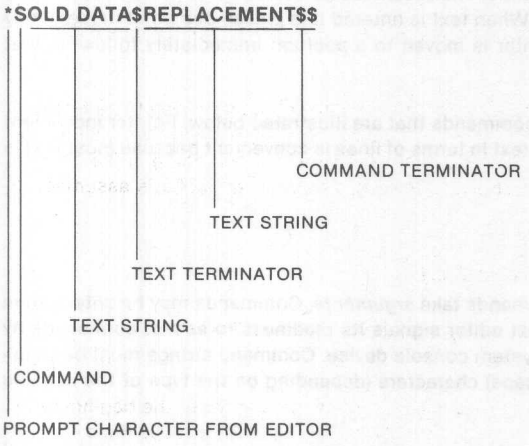
The following three examples show first a typical command, then a command string, and finally a command string combined with a text string. In each case the dollar signs show where the ALT MODE or ESC key was pressed.



This command prints ten lines of text on the system console device.



This command string moves the pointer to the beginning of the text, deletes 20 lines of text, and prints the following five lines on the system console device. The command terminator is placed at the end of the command string; the individual commands do not need terminators.



This command string searches for the string OLD DATA in the text. When found, it is deleted and the text string REPLACEMENT is used as a replacement. The single dollar sign shows where the ESC key was used as the text terminator for OLD DATA; the two dollar signs represent two ESC characters used as the command terminator.

EDITOR COMMANDS

Editor commands are provided to perform four groups of operations: text input/output, pointer manipulation, text modification, and string search and substitution.

B — BEGINNING OF TEXT

The B command moves the pointer to the beginning of the text. It is useful in several ways, such as:

- Defining a starting point when the entire text is to be typed out;
- Moving the pointer to the start of the text prior to starting a search for a selected text string;
- Inserting text at the beginning of the text, ahead of text already there.

Z — END OF TEXT

The Z command positions the pointer immediately following the last character in the text. This command is used mainly to position the pointer so that new text can be inserted after the end of old text.

I — INSERT TEXT

The I command is used to enter text from the system console device, beginning where the pointer is positioned.

Entering a carriage return character causes a line feed character to be generated by the text editor and appended to the carriage return character. Thus, the entry of a carriage return character causes a *pair* of characters to be stored.

After recognizing the letter I as a command, the text editor accepts all subsequent input as text (including carriage returns and appended line feed characters) until ALT MODE, ESC, or Control and C keys are pressed. The ALT MODE or ESC character specifies the termination of the text string; the Control C character cancels the command (and inserts no text).

T — TYPE OUT TEXT

The T command types as many lines of text as the value of the argument written in front of it, as follows:

*nnnnnT$$	nnnnn represents any decimal number from −65,535 to +65,534

If the argument is positive, typing starts at the pointer; the argument value specifies the number of lines to be typed. If the argument is negative, typing begins at the pointer minus the number of lines specified by the argument value; typing continues until the pointer is reached. If the argument value is zero, typing starts at the beginning of the current line; all characters up to the pointer are typed. If no argument value is specified, a default value of 1 is assumed.

EXAMPLES OF EDITING USING B, Z, I, AND T COMMANDS

Suppose you have entered text using the I command, and now wish to type out the entire text. The following command string may be used.

*B500T$$	The B command moves the pointer to the beginning of the text. The 500T command types out 500 lines of text. The argument 500 is assumed to be larger than the number of lines of text; this being the case, the T command is terminated when the end of the text is reached, even though the full count has not been reached. The dollar signs stand for the ALT MODE or ESC character.

Suppose you are entering a source program, using the I command, and have already entered a large number of text lines. For some reason the I command is terminated and the pointer is moved to some other location. When you wish

to resume entering the source file, you simply move the pointer to the end of the text and use the I command again. A typical command string will be as follows:

*ZITEXT STRING----------$$	The new text will be inserted following the old text.

If you are entering text and wish to see the previous five lines, without moving the pointer, the following command may be used:

*-5T$$	The five lines before the current line (the one within which the pointer is located) are printed on the system console device. The current line is printed up to the position of the pointer.

The following command may be used to print the current line of text, without moving the pointer:

*0TT$$	The 0T part of the command prints from the beginning of the line up to the pointer. The following T command prints from the pointer to the end of the line.

L — LINE

The L command moves the pointer as many lines as the value of the argument written in front of it, as follows:

*nnnnnL$$	nnnnn represents any decimal number from – 65,535 to + 65,534

The line feed character serves as the delimiter between lines. A line of text is defined as any text string having a line feed character as its last character. (Recall that a line feed is automatically generated by the text editor when we press carriage return.)

When the argument value is 1 or no argument is used (default value of 1 assumed), the pointer is advanced to the start of the next line. A positive argument value advances the pointer to the beginning of the nth line following the current line. A negative argument value moves the pointer back to the beginning of the nth line preceding the current line. When the argument value is – 1, or just – , the pointer is moved back to the beginning of the line preceding the current line. Finally, if the argument is 0, the pointer is moved to the beginning of the current line.

The command string 0LT finds frequent use. It moves the pointer to the beginning of the current line, and then types the entire line.

K — KILL

The K command deletes as many lines of text as the value of an argument that is placed in front of it, as follows:

*nnnnnK$$	nnnnn represents any decimal number from – 65,535 to + 65,534

A negative argument deletes lines prior to the line containing the pointer. A positive argument deletes lines following the line containing the pointer. If the argument is zero, the characters from the start of the current line up to the buffer pointer are deleted. If the argument is 1 (actual or default), the characters from the pointer, up to and including the line feed character which is used to terminate the line, are deleted.

A — APPEND

The A command reads text from diskette or paper tape into the area in Intellec memory where it is processed. It reads at most 50 lines of text, so it must be issued several times for large files.

F — FIND TEXT STRING

The F command searches for a text string of up to 16 characters. Its format is:

*FXXXXXXXX$ where XXXXXXXX is any text string of up to 16 characters.

The search begins at the pointer, and terminates either upon finding the first occurrence of the string or upon reaching the end of the text without finding a match. If a match is found, the pointer is positioned after the end of the matching string. If a match is not found, the following message is printed:

CANNOT FIND "XXXXXXXX"

BREAK

The string searched for must be terminated in the command with an ALT MODE or ESC character. Other commands may follow in the same string.

S — SUBSTITUTE TEXT STRING

The heavily used S command combines the actions of the Find command with a substitution if a match is found. The format:

*SOLD STRING$REPLACEMENT STRING$

If a match with OLD STRING is found, it is replaced with REPLACEMENT STRING, and the pointer is positioned after the end of the replacement. Each of the strings must be terminated by an ALT MODE or ESC. If no replacement string is included, the old string is simply deleted.

E — EXIT AFTER WRITING TEXT TO OUTPUT FILE

The E command writes text from Intellec memory to diskette or punches it into paper tape for later use. It is ordinarily used at the end of all text editing sessions.

Examples of these commands are found in the sample text editing session that follows.

A SAMPLE TEXT EDITING SESSION

To demonstrate some of the features of the text editor in operation, let us enter a small program and then make some changes to it. The dialog of the session is shown on the facing page with explanatory comments keyed to bold face numbers in the margin.

1. I initialize the system by turning on power and pressing the Reset key on the Intellec front panel. The system responds with the ISIS identification and the ISIS hyphen prompt.

2. I enter the ISIS EDIT command, specifying a file on diskette unit 1 named ONESEC and having an extension of SRC (for source). (Model 210 users enter the editor from the monitor, by typing the command GA800 in response to a dot prompt.)

3. The text editor identifies itself and notes that since there is no file with this name on diskette 1 this is a new file.

4. The editor prompts with an asterisk. Using the I (Insert) command, I enter a simple assembly language program (which will be taken up again in the section on the assembler). The I command continues until I hit ESC twice, as shown by the two dollar signs after the END, which is identifed as step 7.

   In entering the program text I make free use of the tab feature: any time I simultaneously press Control and the letter I, the system responds as a typewriter would to the tab key, with automatic tab stops every eight positions.

5. I type DEALY where I meant DELAY. Noticing the mistake before going on, I press the rubout key three times; the three wrong characters are echoed back as they are erased. I then type the correct characters. To be sure I have the made the correction properly, before hitting carriage return to enter the line I press the Control key and R together, which repeats the line as corrected. Since it appears to be correct, I press carriage return and continue entering the program.

6. I get a line so badly messed up that I decide to start over. Pressing Control and X causes the entire line to be erased; the crosshatch (#) indicates that this was done.

7. I press ESC twice, once to terminate the input string and a second time to terminate the I command.

8. Now there are errors to correct. Using the B command I move the pointer to the beginning of the text, then use the S (substitute) command to correct the spelling of ASSEMBLY. The 0LT combination prints the modified line, which is now correct.

9. I use the F (find) command to locate the label L2, which I entered without the colon. The T (type) command types from where the pointer is positioned after the F; this is to assure myself that I am where I want to be.

10. I insert the colon, then type the entire line.

11. I notice that I have entered an instruction twice, so I use the F to find it.

12. The K (kill) command removes the entire line; to be doubly sure that I removed what I wanted to, I type the three lines before and the three lines after the current position of the pointer.

13. I notice that there is an instruction missing in the subroutine; the missing line should be immediately after the one having the label L3, which means that the pointer must be positioned at the start of the following line, so I use the F command to find the operation code of the following instruction. But there was another JNZ before the one I wanted.

14. I use the L (line) command to move the pointer past the JNZ that I don't want, and use F again. This time the desired instruction is found.

```
1  ISIS-II, V2.2
2  -EDIT :F1:ONESEC.SRC

3  ISIS-II TEXT EDITOR, V1.6
   NEW FILE
4  *I      ; AN ASSEMLBY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
           ; TO THE CONSOLE OUTPUT DEVICE, ONCE EACH SECOND
           ;
   CO      EQU     0F809H  ; PROVIDE ADDRESS OF CO ROUTINE IN MONITOR
           CSEG            ; MAKE THE SEGMENT RELOCATABLE
           STKLN   2       ; SET STACK LENGTH
   START:  LXI     SP,STACK ; INITIALIZE STACK POINTER
   L1:     MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
   L2      MVI     A,200   ; SUBROUTINE PARAMETER
5          CALL    DEALYYLALAY     ; INVOKE SUBROUTINE
           CALL    DELAY   ; INVOKE SUBROUTINE
           DCR     D
           DCR     D
           JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
           ;
           MVI     C,'X'   ; SEND 'X' TO CONSOLE OUTPUT DEVICE
6  CALL    CO      ; MONTIRO CONSOLE#
           CALL    CO      ; MONITOR CONSOLE OUTPUT ROUTINE
           JMP     L1      ; AROIUND THE LOOP INDEFINITELY
           ;
           ; THE DELAY SUBROUTINE
           ;
   DELAY:  MVI     B,12
   L3:     MOV     C,B
           JNZ     L4
           DCR     A
           JNZ     L3
           RET
           ;
           END     START   ; PROGRAM EXECUTION BEGINS WITH SYMBOL 'START'
7  $$
8  *BSLBY$BLY$0LT$$
           ; AN ASSEMBLY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
9  *FL2$T$$
           MVI     A,200   ; SUBROUTINE PARAMETER
10 *I:$0LT$$
   L2:     MVI     A,200   ; SUBROUTINE PARAMETER
11 *FDCR$0LT$$
           DCR     D
12 *K-3T3T$$
   L1:     MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
   L2:     MVI     A,200   ; SUBROUTINE PARAMETER
           CALL    DELAY   ; INVOKE SUBROUTINE
           DCR     D
           JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
           ;
13 *FJNZ$0LT$$
           JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
14 *LFJNZ$0LT$$
           JNZ     L4
```

15. I insert the entire line, including a carriage return, then hit ESC twice.

16. Now I notice an error earlier in the program. I could use L with a negative argument to back up, but the program is short enough that there is no time penalty in simply going back to the beginning and then using an S. (I am reasonably sure that the combination AROIU does not occur elsewhere in the program.)

17. Now I move the pointer to the beginning again and ask for 50 lines to be typed. I don't really know how many lines there are, but certainly less than 50, so I get the entire program.

18. All seems to be in order, so I use the E (exit) command to store the program on diskette (under the name used with the ISIS EDIT command at the beginning), and return to ISIS. (On the Model 210, a paper tape is punched by the E command.)

Why not try it yourself?

Here is a checklist of things you will need to do.

If you have a Model 220 or Model 230:

1. Turn on the Intellec components. Insert an ISIS system diskette in drive 0 and a blank diskette in drive 1.

2. Press the Reset key on the Intellec console and release it. After a brief interval the message

   ISIS-II, Vx.y

   will be produced at the console, where x.y will be numbers indicating the Version number of your ISIS system. (New versions of most programs are issued from time to time.) ISIS will then produce a dash, telling you it is ready to accept a command. Only ISIS prompts with a dash, so any time you see a dash prompt you know you are dealing with ISIS, not the monitor, text editor, ICE, or the Library Manager, which use different prompt characters.

3. Type in the command

   – FORMAT MYDISK.DDM

   Actually you may use any combination of six or fewer characters before the dot and any combination of three or fewer after. What comes after the dot might be your initials or the date or anything else you please.

4. Proceed with the operations shown at the beginning of the text editing session.

5. Save your work on diskette, since the program will be used in Chapter 5.

If you have a Model 210:

1. Turn on the Intellec components.

2. After a brief interval a dot prompt will appear on your console device.

3. Enter the command GA800. This gives control of your Intellec system to the text editor, which in the Model 210 is in a ROM chip set.

4. Proceed with the operations shown at the beginning of the text editing session.

5. If you wish to use the program in connection with the console session on the assembler in Chapter 5, do not turn off the power. Your program will be available when you wish to try the later console session.

As you use your Intellec system you will become very familiar with the text editor, including a few commands that we have not discussed here.

```
15 IL4:     DCR     C
   $$
16 *BSAROIU$AROU$0LT$$
            JMP     L1      ; AROUND THE LOOP INDEFINITELY
17 *B50T$$
            ; AN ASSEMBLY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
            ; TO THE CONSOLE OUTPUT DEVICE, ONCE EACH SECOND
            ;
   CO       EQU     0F809H  ; PROVIDE ADDRESS OF CO ROUTINE IN MONITOR
            CSEG            ; MAKE THE SEGMENT RELOCATABLE
            STKLN   2       ; SET STACK LENGTH
   START:   LXI     SP,STACK ; INITIALIZE STACK POINTER
   L1:      MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
   L2:      MVI     A,200   ; SUBROUTINE PARAMETER
            CALL    DELAY   ; INVOKE SUBROUTINE
            DCR     D
            JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
            ;
            MVI     C,'X'   ; SEND 'X' TO CONSOLE OUTPUT DEVICE
            CALL    CO      ; MONITOR CONSOLE OUTPUT ROUTINE
            JMP     L1      ; AROUND THE LOOP INDEFINITELY
            ;
            ; THE DELAY SUBROUTINE
            ;
   DELAY:   MVI     B,12
   L3:      MOV     C,B
   L4:      DCR     C
            JNZ     L4
            DCR     A
            JNZ     L3
            RET
            ;
            END     START   ; PROGRAM EXECUTION BEGINS WITH SYMBOL 'START'
18 *E$$
```

Chapter 4

THE PL/M LANGUAGE AND COMPILER

The user of microcomputer equipment in the Intel MCS-80 and MCS-85 families has a choice between programming in assembly language, which is somewhat similar to the language of the machine itself, resident FORTRAN-80 (an Intel implementation of ANS FORTRAN 77), or in the high level language called PL/M. In the past the majority of users have relied on assembly language, but increasing numbers are turning to PL/M because of the variety of advantages it offers in many circumstances.

PL/M is a simple language that is easy to learn, understand, and use. It is powerful, in that it gives access to the full power of the microcomputer, but uses a compact notation for expressing the desired processing.

PL/M provides a means of writing a program that:

- often reduces the time and cost of programming
- increases software reliability
- improves documentation
- facilitates software maintenance

All of these advantages are in part a result of the fact that PL/M permits us to write programs that are easy to understand.

PL/M-80 is an Intellec-resident compiler of the PL/M language for the Intel 8080 and 8085 microcomputers.

In this section we shall present only fairly simple examples of programs written in PL/M. Separate tutorial and reference manuals are available if you wish to use the language in your work.*

A PL/M PROGRAM

The simple program that we shall study first is required to read values of two binary variables and produce an output variable that is the "AND" function of the two inputs. For the sake of realism we assume that the inputs describe the status of a door and a switch in some kind of manufacturing process. We assume that these devices have been wired to the inputs of the microcomputer so as to have the following meanings:

For the door:

0 means that the door is closed

1 means that the door is open

*See Appendix II.

For the switch:

0 means that the switch is off

1 means that the switch is on

The specifications say that in order to put the process into operation, the door must be open and the switch must be on. In numeric terms, both values must be one before the process can start.

THE GENERAL CHARACTERISTICS OF A PL/M PROGRAM

Figure 4-1 shows a program that will fulfill these specifications. Let us look first at its overall characteristics before studying the details.

```
/* A PROGRAM THAT ANDS TWO INPUTS TO PRODUCE AN OUTPUT */

ANDER:
DO;
   DECLARE DOOR$1 BYTE;
   DECLARE SWITCH$1 BYTE;
   DECLARE START$PROCESS BYTE;

   DOOR$1 = INPUT(1);
   SWITCH$1 = INPUT(5);
   START$PROCESS = DOOR$1 AND SWITCH$1;
   OUTPUT(1) = START$PROCESS;
END;
```

FIGURE 4-1

We see at the beginning a comment, which is any string of characters that begins with /* and ends with */. We have used a comment here to provide a descriptive heading for the program. Actually, comments may be used anywhere that PL/M permits a space. We occasionally use comments within a program to describe what is being done, when the statements themselves are not obvious. A comment is reproduced verbatim in a program listing but has no effect on the operation of the program.

We see that the program proper begins with a name followed by a colon and that all of the rest of the program is enclosed between the words DO and END. The fact that the entire program is enclosed in the *DO block* defined by this DO-END pair, is signified by indenting all of the statements between the DO-END pair a consistent amount. This indentation is not required by the language and no meaning is derived from it by the computer. It is a very significant aid to human understanding of the program, however.

STATEMENTS

The program consists of a number of *statements*. Every statement ends with a semicolon. For the sake of clarity we never put more than one statement on a line, although the language specifications do permit it. Statements are sometimes written on more than one line, either for clarity or because they are too long to fit on one line.

THE DECLARE STATEMENTS

Every variable used in a PL/M program must appear in a DECLARE statement and all the DECLARE statements in a DO block must appear at the beginning of the block. Each of the DECLAREs here contains just one variable and each is

specified to be a BYTE variable. This means that the representation of the variable within the computer program will be one byte consisting of eight bits. There is also a second type of variable, the ADDRESS type, consisting of 16 bits.

In these DECLARE statements we see that the names of variables, more precisely called *identifiers*, may consist of letters, digits, and dollar signs. The first character of an identifier must be a letter. The dollar sign is used only to improve readability and is ignored by the compiler.

After the DECLARE statements we see a blank line, which we inserted to help clarify the structure of the program, separating the declarations from the statements that specify what processing is to be done.

THE ASSIGNMENT STATEMENT

The statement

```
DOOR$1 = INPUT(1);
```

is an example of an *assignment statement*. The general form of the assignment is

```
variable = expression;
```

The statement means:

> "Evaluate the expression on the righthand side of the equal sign and *assign* that value to the variable written on the lefthand side of the equal sign."

In many cases the expression on the righthand side will be quite simple, as here, but expressions can be more complex.

In the assignment statement just shown, the operation specified is to obtain a byte of data from the input port that has been assigned the number 1. The number of ports available and the identifying numbers associated with them is a matter of system hardware architecture. Not every microcomputer system would necesssarily have ports with the numbers shown here.

The *input* operation obtains a complete byte of eight bits from the port identified by the number within the parentheses. We assume a problem specification which says that the rightmost bit coming in from this port will be either a zero or a one depending on the status of the door. It is also specified that all of the other bits of this byte will be zeros.

Similarly, the byte obtained from input port 5 consists of either a zero or a one in the rightmost bit position and zeros in the other seven bit positions.

THE "AND" OPERATION

Now we come to the statement that is the primary function of the program: to establish whether the door is open and the switch on. This is done with the *logical operator* AND, which operates on all eight pairs of bits of the two bytes specified on the left and right of the AND. For each pair of bits, the result of the AND is a one if both bits are one, and zero otherwise. Thus in the seven bit positions in which each bit of the pair is a zero, the result of the AND will also be a zero. In the rightmost bit position, the result will be a one if the door is open *and* the switch is on; it will be a zero otherwise.

The result of this operation is therefore either a byte of eight zero bits or a byte of seven zero bits and a one. This value, whichever it is, is assigned to the variable having the identifier START$PROCESS.

Finally we send this byte to the output port having the number 1, where we assume that process control hardware has been wired to bit zero of that port, which will start the process. Note that output port 1 has no relation to input port 1.

This completes the action required of this extremely simple program, which would halt operation upon reaching the END statement. No realistic applications program would ever stop in just this way. The language elements required to make the program repeat will be seen in the next example.

COMPILATION

The program shown in Fig. 4-1 is a PL/M *source program*. To be executable by the Intel 8080 or 8085 it must be compiled into an *object program* of machine instructions.

The PL/M compiler is itself a large program. It is called into operation from diskette. When we enter a command such as

```
PLM80 :F1:DOORS.SRC
```

the PL/M compiler is brought into the Intellec system from its storage on the ISIS-II diskette.

THE SOURCE PROGRAM LISTING

One of the outputs of compilation is a listing of the program with certain information added; Fig. 4-2 shows the listing produced when the program of Fig. 4-1 was compiled. We see that the compiler has added a *statement number* in the leftmost position for each statement and a *nesting level indicator* in the next position. The meaning of the latter will become clear in our next example. If there had been errors in the source program, such as missing semicolons at the end of statements, references to variables that had not been declared, or any of a wide variety of errors in the syntax of statements, these errors would have been noted with a brief diagnosis.

```
                /* A PROGRAM THAT ANDS TWO INPUTS TO PRODUCE AN OUTPUT */
1               ANDER:
                DO;
2   1              DECLARE DOOR$1 BYTE;
3   1              DECLARE SWITCH$1 BYTE;
4   1              DECLARE START$PROCESS BYTE;

5   1              DOOR$1 = INPUT(1);
6   1              SWITCH$1 = INPUT(5);
7   1              START$PROCESS = DOOR$1 AND SWITCH$1;
8   1              OUTPUT(1) = START$PROCESS;
9   1           END;
```

FIGURE 4-2

A RAMP FUNCTION

For another example of a simple PL/M program, suppose that we are required to produce what is commonly called a *ramp function*, which in this case outputs a value increasing from 20 to 99 in steps of 1 every half-second. We can imagine that this might be required to increase the speed of a large motor or something of the sort.

The program to do this, shown in Fig. 4-3, introduces two new PL/M features: the *iterative DO* and a *supplied procedure.*

The iterative DO in statement 3 says to increase the value of the variable named CURRENT from 20 to 99; since we did not specify otherwise, an increment of 1 is assumed by default. For each of these values of CURRENT, all of the statements between the iterative DO in statement 3 and its matching END in statement 8 are carried out.

```
                /* A PROGRAM TO PRODUCE A RAMP FUNCTION */

1               RAMP:
                DO;
2     1            DECLARE (CURRENT, COUNTER) BYTE;

3     1            DO CURRENT = 20 TO 99;
4     2               DO COUNTER = 1 TO 25;
5     3                  CALL TIME(200);
6     3               END;
7     2               OUTPUT(1) = CURRENT;
8     2            END;

9     1         END;
```

FIGURE 4-3

The first statement in this range, statement 4, is another iterative DO, this time increasing the variable named COUNTER from 1 to 25. COUNTER is not used otherwise, so the effect is simply to execute statement 5, 25 times.

TIME is the name of a *procedure*, a PL/M procedure being a portion of a program that carries out some specific task when called into action. The procedure TIME is supplied as part of the PL/M system; you can also write your own procedures as part of your program.

The effect of the procedure TIME is to delay program execution for a number of microseconds equal to 100 times the *argument* (the expression written in parentheses). If the argument is 1, the delay is just 100 μs; if the argument is 2, the delay is 200 μs; with an argument of 200, as here, the delay is 20,000 μs. When the procedure is called 25 times as the result of the iterative DO, the total delay is $25 \times 20{,}000\ \mu s = 0.5$ sec.

When this inner DO has been carried out the required 25 times, the value of CURRENT is sent to an output port. Then as a result of the iterative DO in statement 3 the value of CURRENT is increased by 1 and the whole process is repeated.

We see in this program how the nesting level indicators show the *depth of nesting* of statements. Statements 2 and 3, for example, are nested within one DO, statement 4 is nested within two DOs, and statement 5 within three. The END for each DO is given a level number as though included in the range of its matching DO. If the number of DOs and ENDs match, as they must in a correct program, the final END will have a level of 1. Level numbers can be a useful diagnostic tool in finding program errors.

THE RAMP FUNCTION WITH CONSOLE OUTPUT

Let us now see what modifications to this program would be required to make it run on the Intellec system.

The main thing needed is a way to send the value of CURRENT to the consol edevice rather than sending it to an output port. This will require converting the internal representation of the value of CURRENT, which is pure binary, into a decimal number with proper ASCII representation of each digit for use by the external device. We will then need to be able to call upon a procedure to send this number to the console device. Finally, it will be good to provide a way to return control of the system to ISIS when the program has finished its work.

The program in Fig. 4-4 embodies these modifications. The first new feature is the declaration of two procedures that the program will use. The procedure named CO (for console output) takes a BYTE argument and sends the character

```
                /* A PROGRAM TO PRODUCE A RAMP FUNCTION AND SEND RESULTS TO CRT */

  1             RAMP:
                DO;
  2    1           DECLARE (CURRENT, COUNTER) BYTE;

  3    1           CO:
                   PROCEDURE(CHAR) EXTERNAL;
  4    2              DECLARE CHAR BYTE;
  5    2           END;

  6    1           EXIT:
                   PROCEDURE EXTERNAL;
  7    2           END EXIT;


  8    1           DO CURRENT = 20 TO 99;
  9    2              DO COUNTER = 1 TO 25;
 10    3                 CALL TIME(200);
 11    3              END;
 12    2              CALL CO( (CURRENT / 10) OR 30H );     /* TEN'S DIGIT TO CONSOLE OUTPUT (CO) */
 13    2              CALL CO( (CURRENT MOD 10) OR 30H );   /* UNIT'S DIGIT */
 14    2              CALL CO(0DH);                         /* CARRIAGE RETURN */
 15    2           END;

 16    1           CALL EXIT;     /* RETURN TO ISIS */

 17    1        END;
```

FIGURE 4-4

represented by that byte to the console output device. The procedure CO is declared to be EXTERNAL, which means that it is not contained in this program, but must be obtained from some other source. As it happens in this case, the procedure will be obtained from an ISIS library of programs, using the LINK operation; we shall see in the application example in Chapter 7 how such a procedure could also be one that we have programmed, either in PL/M or assembly languge.

The procedure EXIT, also EXTERNAL, simply returns control to ISIS. It does not require an argument.

The two iterative DOs are as before. Now, however, when we want to send the value of CURRENT to the console output device, we must convert from binary to decimal, and also convert the digits to the form the console expects. The conversion of a binary number that has a decimal representation no larger than two digits can be handled as shown. The quotient on division by 10, with the remainder ignored, is the tens digit. We add two bits as required by the ASCII representation of a digit, using the OR operation, and send the digit to the system console using the CO Procedure. The MOD function gives the remainder upon division. As used here, it provides the units digit. A final call of CO sends a carriage return. No line feed is sent, so the two-digit values of CURRENT will all appear in the same place on the CRT screen.

When all the values of CURRENT have been produced, the CALL EXIT returns control of the system to ISIS, which will then issue a prompt, waiting for whatever command we may wish to enter next.

THE LINK AND LOCATE OPERATIONS

This program can be compiled as before. If the name of the file containing this program is

```
:F1:RAMPCO.SRC
```

then the command would be:

```
PLM80 :F1:RAMPCO.SRC
```

The compilation produces two new files, differing from the source file in their extensions. The file

:F1:RAMPCO.LST

contains the *listing file*, which is the form shown in Fig. 4-4. The file

:F1:RAMPCO.OBJ

contains the *object file*, which is a program of 8080 instructions ready to be combined with any other programs that are needed for its execution. In our case, it is necessary to obtain the procedures TIME, CO, and EXIT, which are found in two libraries named SYSTEM.LIB and PLM80.LIB. This can be done with the ISIS command LINK:

LINK :F1:RAMPCO.OBJ, SYSTEM.LIB, PLM80.LIB TO :F1:RAMPCO.LNK

The extension LNK is meant to suggest "link output," but actually you may use any extension you please so long as it is not OBJ. The output of this operation is the file named

:F1:RAMPCO.LNK

which now has all the procedures in it, but still in relocatable form so that we can decide later exactly where it goes in memory. To accomplish this in the simplest way, we use the command:

LOCATE :F1:RAMPCO.LNK

We could specify exactly where we want the program to go in memory, but since we don't really care, for our purposes here, we leave the LOCATE program to pick a location that does not conflict with ISIS functions.

The output of the LOCATE is a final file named

:F1:RAMPCO

i.e., having no extension. We can now type that file name, just as though it were a command — which, in fact, it now is. The program will execute, producing the number from 20 to 99 on the system console, once every half-second.

If you have the PL/M compiler, why not try it? Use the text editor to enter the program, then type the PLM80, LINK, and LOCATE commands as just shown.

SUMMARY

Naturally, there is much more to the PL/M story than it is possible to tell here. Other manuals are available to tell you about these features and how they are useful. Some additional topics are taken up in the PL/M program in Chapter 7.

Chapter 5

THE ASSEMBLERS

Using the Intellec assemblers, you can write programs at the level of the basic microcomputer instructions, and yet gain the advantages of various features that go well beyond the basic machine language, such as:

- Symbolic addressing
- Program relocation, which means that an assembled segment can be placed anywhere in memory (8080/8085 only).
- Macros, which let you tailor a piece of code to multiple uses (ISIS-II assemblers only).
- The INCLUDE facility, by which you can bring pre-written code sections, including macro definitions, from disk into the program (ISIS-II assemblers only).

All Intellec systems come with an 8080/8085 assembler, which we shall use in our examples. Optional assemblers are available for the MCS-48 family of microprocessors. Not all assemblers have all the features discussed here.

In this section we shall sketch some of these facilities in terms of a simple illustrative program that is related to the last PL/M example in the previous chapter, and parts of which will be utilized in the application example in Chapter 7.

It is not the intention here to teach assembly language programming or to provide a tutorial on microprocessor instructions and organization. The reader not familiar with these matters is not expected to be able to understand the details of the illustrative program.

AN ASSEMBLY LANGUAGE PROGRAM TO SEND OUTPUT TO THE SYSTEM CONSOLE

The simple program that we shall study is required only to send the letter 'X' to the system console, once every second. This is about the simplest program that has any output, but it will let us see many of the features of the assembler in action. A second version will bring out the essentials of macros.

A listing of the program is shown in Fig. 5-1. You may recognize it, since it was the example in the illustrative monitor and text editing sessions. This version is the listing produced by the assembler, which contains the assembled machine language instructions and line numbers, as well as everything that was in the source program as we entered it. We shall study the program from this listing version; it will be more meaningful to discuss the assembly process after that.

The headings above the program are: LOC is the location where the assembled instruction would be loaded if the program started at location zero; OBJ is the assembled object program instruction; SEQ is the sequence number, which

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0         MODULE   PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1          ; AN ASSEMBLY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
                      2          ; TO THE CONSOLE OUTPUT DEVICE, ONCE EACH SECOND
                      3          ;
  F809                4 CO       EQU     0F809H  ; PROVIDE ADDRESS OF CO ROUTINE IN MONITOR
                      5          CSEG            ; MAKE THE SEGMENT RELOCATABLE
                      6          STKLN   2       ; SET STACK LENGTH
  0000 310000   S     7 START:   LXI     SP,STACK ; INITIALIZE STACK POINTER
  0003 1632           8 L1:      MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
  0005 3EC8           9 L2:      MVI     A,200   ; SUBROUTINE PARAMETER
  0007 CD1600   C    10          CALL    DELAY   ; INVOKE SUBROUTINE
  000A 15            11          DCR     D
  000B C20500   C    12          JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
                     13          ;
  000E 0E58          14          MVI     C,'X'   ; SEND 'X' TO CONSOLE OUTPUT DEVICE
  0010 CD09F8        15          CALL    CO      ; MONITOR CONSOLE OUTPUT ROUTINE
  0013 C30300   C    16          JMP     L1      ; AROUND THE LOOP INDEFINITELY
                     17          ;
                     18          ; THE DELAY SUBROUTINE
                     19          ;
  0016 060C          20 DELAY:  MVI     B,12
  0018 48            21 L3:     MOV     C,B
  0019 0D            22 L4:     DCR     C
  001A C21900   C    23          JNZ     L4
  001D 3D            24          DCR     A
  001E C21800   C    25          JNZ     L3
  0021 C9            26          RET
                     27          ;
  0000          C    28          END     START   ; PROGRAM EXECUTION BEGINS WITH SYMBOL 'START'

PUBLIC SYMBOLS

EXTERNAL SYMBOLS

USER SYMBOLS
CO     A F809    DELAY  C 0016    L1     C 0003    L2     C 0005    L3     C 0018    L4     C 0019    START  C 0000

ASSEMBLY COMPLETE,  NO ERRORS
```

FIGURE 5-1

we more commonly call the line number; SOURCE STATEMENT is what we wrote, reproduced exactly from the source program.

On every line, anything after a semicolon is taken as a comment. It is carried along to the listing, but has no effect on the assembly. Thus, any line that begins with a semicolon is entirely a comment. We see that no code is ever generated for such a line.

Line 4, the first one that has any effect on the assembler, establishes a meaning for the symbol CO, which is the entry point for the Console Output routine in the monitor. With the symbol CO equated to the hex address 0F809, any appearance of that symbol in the program will be replaced with the numerical address. We see that this has been done in the CALL instruction on line 15.

Line 5 says that this entire program is to be a *code segment* (CSEG) which means that it can be relocated. In brief, it will be possible, using the LOCATE command, to put the assembled program anywhere in memory that we please. This is very useful in product development, when we may not know — as the program is being written — where the different segments will fit — or, indeed, even how much memory there will be.

Line 6 is a final preliminary instruction to the assembler, this time giving the maximum length of the 8080 stack so that appropriate memory space can be allocated. In this extremely simple program, the stack is never more than two bytes in length, so we specify that stack length.

Line 7 is the first instruction. In its label field we see START; we will be able later (line 28) to refer to this location symbolically without having any idea now what the actual memory location of the instruction may turn out to be. The operation code LXI means Load Register Pair Immediate, and in the operand field we see that the stack pointer is

being loaded with the stack origin address using the reserved word STACK. Following the semicolon, a comment explains the purpose of the instruction.

The instructions on lines 8 and 9 also have labels since we need to refer to them from elsewhere in the program, but the CALL instruction in line 10 does not. The CALL invokes the subroutine named DELAY, which starts in line 20, transferring control to that location after placing on the stack the information necessary for the RET (return) instruction at the end of the subroutine to get back to the instruction after the CALL.

The rest of the program follows similar patterns.

The last line of the program is an END, which must always be present to inform the assembler that nothing else follows. The START in its operand field causes program execution to begin with the instruction having that label, when the program is loaded.

THE ASSEMBLY PROCESS

When we began editing the program, we gave it the name :F1:ONESEC.SRC. The :F1: specifies diskette unit 1; ONESEC is the name of the file; SRC is the extension, which stands for source. When we want to assemble the program, we use the ISIS command for the assembler we want, ASM80, ASM48, etc., depending on the microprocessor on which the program will run. We might use the command

```
ASM80 :F1:ONESEC.SRC
```

There are two output files from this process, both on the same diskette as the source program, and both having the same name, but different extensions. The file :F1:ONESEC.LST is the listing file, which is what was shown in Fig. 5-1. On the left side of this listing are the machine instructions assembled from the source program, in relocatable form since we made a CSEG (code segment) of the program. The instructions that will have to be modified when the program is relocated are marked with a C.

Users of the Intellec Series II Model 210 call the assembler into operation by typing the command GB800 in response to a dot prompt from the monitor. The Model 210 assembler produces absolute code and does not have the macro facility.

RELOCATION

The second file that results from assembly is :F1:ONESEC.OBJ which contains the object program instructions — essentially what we see on the left side of the listing, formatted to be acceptable by the LOCATE program (and the LINK program, as we shall consider later). In other words, the output of assembly is called an object program, but it is not *quite* ready for execution. (Assembly language programs can be written that do not require use of the LOCATE program, but only at the expense of deciding at the time of assembly the absolute location in main storage where they should be loaded, which is seldom desirable.)

Converting the relocatable object program into an absolute version ready to be executed is the function of the ISIS program called LOCATE. In our case we might enter the ISIS command

```
LOCATE :F1:ONESEC.OBJ CODE (4000H)
```

This identifies the file to be processed, and specifies that the program is to be prepared for loading into absolute memory location 4000 hexadecimal. If needed, we could also specify the absolute location of the stack, the data (of which we have none in this program), and the free memory area (ditto). The output of this command is a final file, identified as

```
:F1:ONESEC
```

i.e., it has no extension. This file is now ready to be executed simply by entering its name, which makes it, in effect, a command.

Why not try it? If you entered the program while studying the text editor section, all you need do is enter the commands just described.

Here are the steps to follow in running the program.

If you have a Model 220 or 230:

1. Turn on the Intellec components. Insert an ISIS system diskette in drive 0 and the diskette containing the source program in drive 1.

2. Press the Reset key on the Intellec front panel and release it.

3. ISIS will respond with a sign-on message and a dash prompt to indicate that it is ready to execute commands.

4. Enter the command

   ```
   ASM80 :F1:ONESEC.SRC
   ```

   or whatever other file name you have used.

5. You can get a program listing — if you have a printer — with the command

   ```
   COPY :F1:ONESEC.LST TO :LP:
   ```

   If you do not have a printer, you can use the console output device to study the listing, with

   ```
   COPY :F1:ONESEC.LST TO :CO:
   ```

   Output to the CRT will be much too fast to read; you can stop the output by pressing Control and S, and start it again by pressing Control and Q.

6. To correct errors in the program, if any are reported by the assembler, edit the program using

   ```
   EDIT :F1:ONESEC.SRC
   ```

   Remember to start with an Append command to the text editor, and remember to terminate the text editing session with an Exit to get the modified program back on diskette.

7. After you have a correct assembly (no errors reported) execute the command

   ```
   LOCATE :F1:ONESEC.OBJ CODE(4000H)
   ```

8. Now execute the command

   ```
   :F1:ONESEC
   ```

   X's should start appearing on your system console. If not, there is most likely a typing error in the source program that happens not to be diagnosable by the assembler. Check the listing carefully.

9. To return control to ISIS, press the Interrupt 1 key on the Intellec front panel.

10. Remove the diskettes and turn off the power.

If you have a Model 210:

1. Turn on the Intellec components.
2. Press the Reset key on the Intellec front panel and release it. The monitor will respond with a dot prompt.

   Naturally, if you left the power on from the text editing session in Chapter 3, you omit these steps.
3. Type the command GA800 to enter the text editor.
4. Make the following changes in the program:

   a. Replace the CSEG in line 5 with the instruction

      ORG 1400

   b. Delete line 6.

   c. In line 7, change STACK to 1422.
5. To assemble the program, enter the command GB800. The assembler, after a brief period, will type the characters P=. Type a 1 and press carriage return. When the characters P= are typed again, enter 2. The third time, enter 3. (These are the three passes of your assembler, which in the Model 210 is in a ROM chip set.)
6. To correct any errors in the program, which will be reported to your console device, enter the text editor again by typing GA800, and reassemble.
7. Enter the command G1400 to begin execution of your program. X's should start appearing on your system console. If not, there is most likely a typing error in the source program that happens not to be diagnosable by the assembler. Check the listing carefully, then use the text editor to correct any errors and reassemble.
8. To return control to the monitor, press the Interrupt 0 key on the Intellec front panel.
9. Turn off the power.

THE MACRO ASSEMBLER

Suppose now that we expand the requirement on the program, so that the amount of delay between characters is variable within certain limits and so that it is to be convenient to send *any* character to the CRT, rather than just the letter X. What we wish to do, specifically, is to send the numeral zero to the system console with a 1.5 second delay, followed by a plus sign after a 0.25 second delay, and then repeat these two operations indefinitely.

Naturally, we could write a program to do all this using the same techniques as before. However, there is a better solution, especially if there were a need for further flexibility in these operations. The solution is to make a *macro* of the code needed to generate a given time delay, which means that a simple symbol — the *marco name* — stands for a complete group of instructions. The source program in Fig. 5-2 shows how this is done, using the ISIS macro assembler. (The listing is shown later.)

First of all, it is the *same* assembler, which we invoke with the same command (ASM80, ASM48, etc.) as before. However, to tell the assembler that we will have macros in the program, we begin with the line $MACROFILE. The dollar sign identifies this as an *assembler control*. (There are others.) Use of $MACROFILE requires 48K bytes of Intellec memory.

The program begins with the CSEG operation, as before, but this time we are handling the symbol CO differently, identifying it as external (EXTRN). This tells the assembler that the value of this symbol will be supplied later, in the LINK operation.

```
$MACROFILE
        ; MACRO VERSION OF THE PROGRAM TO SEND 'X' TO CONSOLE OUTPUT
        ; DEVICE, ONCE EACH SECOND.
        ; THIS VERSION MAKES THE DELAY VARIABLE, AND SENDS ANY CHARACTER.
        ;
        CSEG            ; MAKE THE SEGMENT RELOCATABLE
        EXTRN   CO      ; CONSOLE OUTPUT ADDRESS SUPPLIED IN LINK
        STKLN   2       ; SET STACK LENGTH
        ;
TIME    MACRO   T1,T2
        LOCAL   L1
        MVI     D,T1    ; LOAD FIRST PARAMETER INTO D REG
L1:     MVI     A,T2    ; LOAD SECOND PARAMETER INTO A REG
        CALL    DELAY   ; INVOKE SUBROUTINE
        DCR     D
        JNZ     L1
        ENDM
        ;
        ; THIS ENDS THE MACRO DEFINITION - NOW COMES THE PROGRAM
        ;
START:  LXI     SP,STACK ; INITIALIZE THE STACK POINTER
L1:     TIME    100,150 ; MACRO REFERENCE - 1.5 SEC
        MVI     C,'0'   ; SEND ZERO TO CONSOLE OUTPUT DEVICE
        CALL    CO      ; CONSOLE OUTPUT ROUTINE IN MONITOR
        TIME    50,50   ; MACRO REFERENCE - 0.25 SEC
        MVI     C,'+'   ; SEND PLUS SIGN TO CONSOLE OUTPUT DEVICE
        CALL    CO
        JMP     L1      ; AROUND THE LOOP INDEFINITELY
        ;
        ; THE DELAY SUBROUTINE
        ;
DELAY:  MVI     B,12
L3:     MOV     C,B
L4:     DCR     C
        JNZ     L4
        DCR     A
        JNZ     L3
        RET
        ;
        END     START   ; PROGRAM EXECUTION BEGINS AT LABEL 'START'
```

FIGURE 5-2

After giving the stack length information, which is as before, we have the definition of the macro named TIME. The code defined in the macro utilizes the subroutine named DELAY, but with two parameters that make the delay variable.

The amount of the delay is controlled by the immediate data in the two MVI instructions. The DELAY subroutine delays a number of microseconds equal to 100 times the value of the number in the A register when it is called. The program is arranged to call DELAY repeatedly. To be precise, and describing it in terms of the parameters used in the MACRO assembler directive, we wish to call the DELAY subroutine T1 times, and place in the A register the number T2 when it is called.

The assembler directive MACRO tells the assembler that what follows is not to be assembled, but rather is a *definition* of what is to be assembled when the macro is later referenced. We see that in the two MVI instructions there are no actual numbers, but only the parameters T1 and T2.

Since this macro will be referenced twice in the executable instructions that follow, we could face a problem with the label L1, which the assembler would reject as multiply-defined. To prevent this, we have the pseudo operation LOCAL, which tells the macro assembler that the label L1 has separate meaning within each reference to TIME.

The ENDM (end macro) assembler directive tells the assembler that we have completed the definition of the macro.

None of what we have discussed so far results in the generation of any code. This is only the macro *definition*. Now we come to the program itself, in which this definition is referenced. The line with the label L1 is one such: it references the macro TIME, specifying that for the parameter T1 the assembler should substitute 100, and that for the parameter T2 it should substitute 150. Now we are ready to place the desired character in the C register and call CO. Next comes another reference to TIME, with different parameters.

As with the earlier version, the program loops indefinitely.

The DELAY subroutine is as before.

The listing in Fig. 5-3 was produced from this program by the assembler. Look at line 22, which is the first reference to the macro TIME. We see that there is no code on this line, but following it are the instructions of the macro definition, "particularized" with the parameters specified in the reference. All instructions generated from the macro have plus signs after the line number. We see that 100 has been substituted for T1 and 150 for T2. The L1 from the macro definition has become ??0001, which is treated as entirely different from the symbol L1 in line 22 and from the symbol ??0002 in line 32.

Looking at line 30 we see that the same macro as before, TIME, now has different parameters, which have been properly substituted into the instructions.

If you wish, you may use the text editor to modify the earlier program along these lines. Either enter the entire program, with a new name, or modify the earlier program. In the process, you may as well as use different delays and characters, or expand the program to send a variety of characters to the system console. Your only limits are that you cannot go faster than your system console can accept characters, and the largest values you may use for the two parameters in TIME are 255, since these are BYTE values that must fit in MVI instructions.

After you have assembled the program successfully, it will be necessary to obtain the value of the symbol CO, which was declared in the program to be external. This is what the ISIS LINK program does for us, using the following command with a new file named ONEMAC:

```
LINK :F1:ONEMAC.OBJ, SYSTEM.LIB TO :F1:ONEMAC.LNK
```

As noted before, LNK is not a required extension; you may use any extension you please so long as it is not OBJ.

Now the LOCATE operation must be applied to the linkage output:

```
LOCATE :F1:ONEMAC.LNK
```

The result is a program named :F1:ONEMAC (no extension), which you can run by entering its name when ISIS indicates — by typing a hyphen prompt — that it is ready to accept a command.

THE INCLUDE FACILITY

It can happen in writing programs that a piece of code turns up in many different situations, making it desirable to be able to write the code, test it, and then place it on diskette for use whenever needed. This is what the INCLUDE facility provides.

Whenever we want a piece of code inserted into the program, we write the line

```
$INCLUDE filename
```

giving the name under which the code was stored on diskette.

```
ASM80 :F1:ONEMAC.SRC DEBUG

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0          MODULE   PAGE    1

  LOC  OBJ              SEQ        SOURCE STATEMENT

                          1 $MACROFILE
                          2         ; MACRO VERSION OF THE PROGRAM TO SEND 'X' TO CONSOLE OUTPUT
                          3         ; DEVICE, ONCE EACH SECOND.
                          4         ; THIS VERSION MAKES THE DELAY VARIABLE, AND SENDS ANY CHARACTER.
                          5         ;
                          6         CSEG              ; MAKE THE SEGMENT RELOCATABLE
                          7         EXTRN   CO        ; CONSOLE OUTPUT ADDRESS SUPPLIED IN LINK
                          8         STKLN   2         ; SET STACK LENGTH
                          9         ;
                         10 TIME    MACRO   T1,T2
                         11         LOCAL   L1
                         12         MVI     D,T1      ; LOAD FIRST PARAMETER INTO D REG
                         13 L1:     MVI     A,T2      ; LOAD SECOND PARAMETER INTO A REG
                         14         CALL    DELAY     ; INVOKE SUBROUTINE
                         15         DCR     D
                         16         JNZ     L1
                         17         ENDM
                         18         ;
                         19         ; THIS ENDS THE MACRO DEFINITION - NOW COMES THE PROGRAM
                         20         ;
0000 310000      S       21 START:  LXI     SP,STACK ; INITIALIZE THE STACK POINTER
                         22 L1:     TIME    100,150 ; MACRO REFERENCE - 1.5 SEC
0003 1664                23+        MVI     D,100     ; LOAD FIRST PARAMETER INTO D REG
0005 3E96                24+??0001: MVI     A,150     ; LOAD SECOND PARAMETER INTO A REG
0007 CD2600      C       25+        CALL    DELAY     ; INVOKE SUBROUTINE
000A 15                  26+        DCR     D
000B C20500      C       27+        JNZ     ??0001
000E 0E30                28         MVI     C,'0'     ; SEND ZERO TO CONSOLE OUTPUT DEVICE
0010 CD0000      E       29         CALL    CO        ; CONSOLE OUTPUT ROUTINE IN MONITOR
                         30         TIME    50,50     ; MACRO REFERENCE - 0.25 SEC
0013 1632                31+        MVI     D,50      ; LOAD FIRST PARAMETER INTO D REG
0015 3E32                32+??0002: MVI     A,50      ; LOAD SECOND PARAMETER INTO A REG
0017 CD2600      C       33+        CALL    DELAY     ; INVOKE SUBROUTINE
001A 15                  34+        DCR     D
001B C21500      C       35+        JNZ     ??0002
001E 0E2B                36         MVI     C,'+'     ; SEND PLUS SIGN TO CONSOLE OUTPUT DEVICE
0020 CD0000      E       37         CALL    CO
0023 C30300      C       38         JMP     L1        ; AROUND THE LOOP INDEFINITELY
                         39         ;
                         40         ; THE DELAY SUBROUTINE
                         41         ;
0026 060C                42 DELAY:  MVI     B,12
0028 48                  43 L3:     MOV     C,B
0029 0D                  44 L4:     DCR     C
002A C22900      C       45         JNZ     L4
002D 3D                  46         DCR     A
002E C22800      C       47         JNZ     L3
0031 C9                  48         RET
                         49         ;
0000             C       50         END     START     ; PROGRAM EXECUTION BEGINS AT LABEL 'START'
```

FIGURE 5-3

Suppose, for example, that we are writing a series of programs, many of which require a time delay. We could place the TIME macro on diskette, giving it the name

:F1:TIMEM.SRC

(the name doesn't really matter — it doesn't have to be the same as the macro although it may be). Then, instead of actually typing in the macro using the text editor — which is time consuming and error prone — we instead type in the line

$INCLUDE :F1:TIMEM.SRC

and the job is done.

The INCLUDE facility is used to obtain source code for inclusion in a PL/M or assembly language program, prior to compilation or assembly. This is distinguished from the Library Manager, which is used to obtain object code segments during the LINK operation.

CONCLUSION

As noted at the beginning of this section, it isn't possible (or really appropriate) to try to give the whole story on the assembler in a manual of this type. When you are ready, the manuals are there. (See the Guide to Other Manuals at the back of the book.) We trust this sketch has given you some idea of the power of the ISIS macro assembler, and whetted your appetite to know more.

Chapter 6

IN-CIRCUIT EMULATION

WHAT IS IN-CIRCUIT EMULATION?

In-circuit emulation is a tool for developing products designed around the Intel 8080, 8085, 8048, and Series 3000 CPUs. Its flexible debugging commands and resource lending allow you to diagnose your prototype system's hardware and software in the earliest stages of development, even before a prototype is built, or while the prototype is in development and its memory and input/output (I/O) facilities are incomplete.

In-circuit emulation allows you to perform this diagnosis, for the most part, through console dialogue. The programmer needs no separate software environment, such as simulation with a time-shared computer; the engineer needs no laboratory model equipped with specially-built diagnostic aids.

Not all of the features in the following general description are available in every ICE system.

IN-CIRCUIT EMULATION HARDWARE

The in-circuit emulation hardware consists of printed circuit boards that are inserted into the Intellec cabinet. In-circuit emulation can be used as a software development tool prior to the building of a prototype. If a prototype exists, an umbilical cable connects the in-circuit emulation circuitry in the Intellec cabinet to your prototype. A 40-pin plug at the end of this cable plugs into your prototype system in place of its CPU, allowing the in-circuit emulation hardware to emulate all the functions of the CPU. This connection between ICE and a prototype is shown in Fig. 6-1.

IN-CIRCUIT EMULATION SOFTWARE

The in-circuit emulation software commands are your interface to in-circuit emulation. These are easy-to-use commands that you enter from the Intellec console to communicate with the in-circuit emulation hardware. The in-circuit emulation software is loaded into Intellec memory from diskette. We shall survey the function of the in-circuit emulation software shortly.

WHAT DOES IN-CIRCUIT EMULATION DO FOR YOU?

In-circuit emulation can help you develop your prototype system in two ways. First, it provides a wide range of diagnostic commands, enabling you to debug your prototype hardware and software. Second, it lets you "borrow" the physical resources of the Intellec system, and use them as if they were resident in your prototype system, until such time as your own prototype is complete.

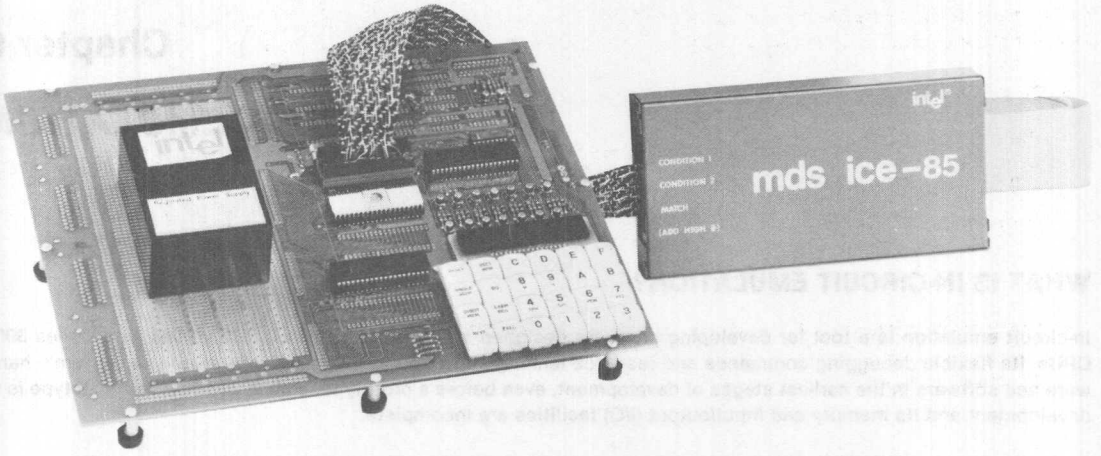


FIGURE 6-1

RESOURCE LENDING

The in-circuit emulation hardware 40-pin connection to your prototype system allows it to emulate the prototype's CPU. Also, some or all of your prototype system's storage can be emulated by Intellec memory. Similarly, you don't need to attach special debugging peripherals to your prototype, because with in-circuit emulation your prototype may share all Intellec peripherals.

In-circuit emulation has an address mapping facility (described later) for handling the addressing of borrowed resources. Whenever the emulated CPU makes a memory or I/O port reference, in-circuit emulation first consults its address map to determine whether the specified memory exists and, if so, whether it is in the prototype or is being borrowed from the Intellec system. In the latter case any needed address translation is made.

One of your first tasks in an in-circuit emulation debugging session is to set up memory and I/O port address mapping in accordance with the needs of your software. This is done using the in-circuit emulation mapping command, which allows you to define and display memory and I/O port mapping.

DEBUGGING

Prototype system software can be loaded into borrowed memory, prototype memory, or any combination of the two, and run as if it were resident in the prototype, thus emulating your eventual production system. You can perform this emulation at real-time speed (to check out prototype system timing, for example) or in single- or multiple-instruction steps. You can stop emulation manually at any time to examine system status, or you can specify "break conditions" as part of the emulation command. Emulation halts automatically when a break condition is satisfied, after which you can have in-circuit emulation perform additional operations and resume emulation automatically if you like.

Break conditions can be specified in many ways. For example, you can specify that emulation is to continue until a register contains a specified value, or until a given memory location is read or written, or there is an input or output operation for a particular port, or the instruction at a given location is executed. There are many other stopping conditions, leading to a highly flexible debugging package.

In-circuit emulation debugging power can also be seen in the commands for changing and displaying information describing the state of your prototype system. The specific items defining the state are:

- CPU pins, registers, and flags
- The contents of the stack
- The contents of memory
- The contents of I/O ports

In addition, you can request information on:

- The address or the operation code of the last instruction emulated
- A trace of the most recently emulated instructions, machine cycles, or 18 lines of system status data of your choice.
- Current subroutine nesting

Remember, the CPU being interrogated is, in effect, the CPU of your prototype system. In-circuit emulation lets you emulate all of your CPU functions, even though your CPU is not installed, and even if your prototype has not been built yet. In short, it is a powerful development tool for debugging hardware and software, at any stage of the development process for your microcomputer-based product.

SYMBOLIC DEBUGGING

One of the most important features of in-circuit emulation is that you can refer to symbols and locations in a program entirely in symbolic terms. You don't have to know where a variable is located, for example, or where the code corresponding to a PL/M line number or assembler label is; you simply use the symbols directly. For example, we can issue the ICE-85 command

```
GO TILL .CYCLE EXECUTED
```

Assuming that CYCLE is a label or PL/M procedure name, emulation continues until the instruction at that location has been executed. Or we could say

```
GO TILL .CARS$WAITING WRITTEN
```

which says to stop emulation when the program stores anything at the location corresponding to the PL/M symbol CARS$WAITING.

PL/M line numbers can also be referred to:

```
GO FROM .START TILL #56 EXECUTED
```

This says to begin emulation with the instruction at the symbolic location START, and continue until the instruction corresponding to the PL/M line number 56 has been executed.

Actual machine locations can of course also be used when more convenient, but symbolic debugging offers such power and flexibility that it will much more commonly be employed. Further examples will be found in the console sessions at the end of this chapter and at the end of Chapter 7.

A SKETCH OF THE IN-CIRCUIT EMULATION COMMANDS

Entering in-circuit emulation is a simple matter of typing the ISIS command

ICE85 (or ICE80 or ICE48 or whatever system you have).

The ICE system prompts with an asterisk.

Once the system is under the control of the ICE software, you may use any command by typing in its name or abbreviation. For an overview of these commands, they may be divided into three categories: emulation, interrogation, and utility.

In the rest of this material, and in the terminal session at the end of the chapter, we shall describe in-circuit emulation in terms of ICE-85.

EMULATION COMMANDS

Using the ICE-85 GO command, you can emulate the prototype system at real-time speed. Using the STEP command, emulation can be performed in single- or multiple-instruction steps. In either case, emulation of the prototype program can be started at any instruction.

The GO command allows you to specify up to two break conditions for halting emulation or emitting an oscilloscope synchronization pulse when the condition occurs. A break condition can be satisfied by executing a specified instruction (location or operation code), by referencing a specified location or I/O port (read or write), or by referencing a specified data value. The ICE-85 hardware provides you with an additional break capability, an 18-channel external trace module. This causes a break whenever a specified system signal or combination of signals changes. The STEP command breaks after a specified number of instructions have been executed or a condition is satisfied.

Emulation commands are summarized in Table 6-1.

TABLE 6-1. ICE-85 EMULATION COMMANDS

ICE-85 COMMAND	DESCRIPTION
GO	Causes emulation to run until a break condition is satisfied.
STEP	Causes emulation to run for a specified number of instructions or until a software condition is satisfied.
CALL	Permits emulation of interrupt service routines.

INTERROGATION COMMANDS

With ICE-85 at command level following a break in emulation, you can display the items listed earlier and alter the contents of some of these items. The ICE-85 commands used in these interrogation operations are summarized in Table 6-2.

UTILITY COMMANDS

The ICE-85 command language also provides for various utility operations such as loading or saving program files, defining symbols, or returning to ISIS control. These utility commands are summarized in Table 6-3.

TABLE 6-2. ICE-85 INTERROGATION COMMANDS

ICE-85 COMMAND	DESCRIPTION
BASE	Establishes base of displayed data (decimal, octal, hexadecimal, etc.).
SUFFIX	Establishes default base for numbers entered from the console.
Display	Various items can be displayed by typing in their names.
Change	Contents of a specified item can be altered by entering its name followed by a value.
SEARCH	Causes memory locations containing specified contents to be displayed.
PRINT	Causes trace information to be displayed.

TABLE 6-3. ICE-85 UTILITY COMMANDS

ICE-85 COMMAND	DESCRIPTION
LOAD	Fetches object code and symbol table from input medium.
SAVE	Causes symbol table and object code to be reproduced on output medium.
LIST	Specifies list file used for output of terminal dialog.
DEFINE	Enters additional symbols and their values into symbol table.
EXIT	Causes program control to return to ISIS.

MEMORY MAPPING

ICE-85 commands can reference:

- Prototype system memory.
- Intellec memory substituting for prototype system memory.

To understand how ICE-85 distinguishes among these two, we must first define the terms *logical* and *physical* as used in the in-circuit emulation context. For the sake of illustration, suppose I send an annual Christmas card to an old friend at his 1000 H Street address. Last year he moved to 3000 H Street without telling me, so I continue addressing my cards to his former location. *Logically*, my friend still resides at 1000 H, but *physically* he is located at 3000 H. This presents no problem as long as my Christmas card finds its way through the post office's letter-forwarding (address-mapping) mechanism.

Similarly, one can map (logical) program addresses into physical Intellec memory locations. The addresses have not changed as far as the program knows, and ICE-85 will take care of the address mapping details. Using ICE-85's MAP command, you can specify prototype memory and I/O ports to be *logically nonexistent* (GUARDED), *logically and physically existing in the prototype system's memory* (USER), or *logically existing in your system while physically existing in Intellec memory* (INTELLEC). The initial setting for all blocks is GUARDED. Any reference to a guarded memory location or I/O port causes an error to be issued; this is a useful software debugging feature.

Logical addresses in your program may range from 0 to 65,535 (64K) and are partitioned into 32 blocks of 2K bytes. Each logical memory block can reside physically in your own system or in any unused 2K block of Intellec memory.

I/O PORT MAPPING

Logical I/O ports range from 0 to 255 and are partitioned into 32 blocks of 8 ports each. Each block of logical I/O ports can reside physically in your own or the Intellec system. Unlike memory, I/O port numbers cannot be altered when they are mapped physically into the Intellec memory. The physical I/O port number must be the same as the logical I/O port being mapped.

PRODUCT DEVELOPMENT SEQUENCE

The interaction between you as the designer of a new product, the Intellec system, and in-circuit emulation can be summarized in the following product development sequence:

1. The prototype system design specification and initial software and hardware design are completed.
2. Prototype software is written in PL/M, Fortran, or assembly language and an object file is prepared.
3. Using in-circuit emulation, program debugging can begin, with memory and I/O resources borrowed from the Intellec system, even though no prototype hardware exists.
4. Skeleton prototype hardware is built. The skeleton must include as a minimum a socket for the CPU and a user bus.
5. The in-circuit emulation umbilical cable is attached to your prototype equipment. The in-circuit emulation software is loaded into memory.
6. Peripherals and memory not present in your skeleton prototype are mapped into the Intellec system.
7. Prototype hardware and software are exercised with the in-circuit emulation commands.
8. Memory, I/O, and peripheral hardware are added to the prototype system as they are developed. The corresponding resources are no longer "borrowed" from the Intellec system.
9. The in-circuit emulation umbilical cable is unplugged and replaced with your own CPU chip.

At no point in this development sequence have you had to provide extraneous hardware diagnostic equipment or specialized software debugging aids for prototype system development.

AN IN-CIRCUIT EMULATION TERMINAL SESSION

As we have done before with the monitor and the text editor, we shall tie these ideas together with an illustrative terminal session. The vehicle will be the assembly language program of Chapter 5 to send the letter X to the console output device, once per second, plus a modified version that contains a coding error. The version we shall work with first is shown in Fig. 6-2; this is identical to Fig. 5-1.

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0         MODULE   PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1          ; AN ASSEMBLY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
                      2          ; TO THE CONSOLE OUTPUT DEVICE, ONCE EACH SECOND
                      3          ;
  F809                4 CO       EQU     0F809H  ; PROVIDE ADDRESS OF CO ROUTINE IN MONITOR
                      5          CSEG            ; MAKE THE SEGMENT RELOCATABLE
                      6          STKLN   2       ; SET STACK LENGTH
  0000 310000   S     7 START:   LXI     SP,STACK ; INITIALIZE STACK POINTER
  0003 1632           8 L1:      MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
  0005 3EC8           9 L2:      MVI     A,200   ; SUBROUTINE PARAMETER
  0007 CD1600   C    10          CALL    DELAY   ; INVOKE SUBROUTINE
  000A 15            11          DCR     D
  000B C20500   C    12          JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
                     13          ;
  000E 0E58          14          MVI     C,'X'   ; SEND 'X' TO CONSOLE OUTPUT DEVICE
  0010 CD09F8        15          CALL    CO      ; MONITOR CONSOLE OUTPUT ROUTINE
  0013 C30300   C    16          JMP     L1      ; AROUND THE LOOP INDEFINITELY
                     17          ;
                     18          ; THE DELAY SUBROUTINE
                     19          ;
  0016 060C          20 DELAY:  MVI     B,12
  0018 48            21 L3:     MOV     C,B
  0019 0D            22 L4:     DCR     C
  001A C21900   C    23          JNZ     L4
  001D 3D            24          DCR     A
  001E C21800   C    25          JNZ     L3
  0021 C9            26          RET
                     27          ;
  0000          C    28          END     START   ; PROGRAM EXECUTION BEGINS WITH SYMBOL 'START'

PUBLIC SYMBOLS


EXTERNAL SYMBOLS


USER SYMBOLS
CO     A F809    DELAY  C 0016    L1     C 0003    L2     C 0005    L3     C 0018    L4     C 0019    START  C 0000

ASSEMBLY COMPLETE,  NO ERRORS
```

FIGURE 6-2

We shall see the central concepts of In-Circuit Emulation, including storing the program in the memory of prototype hardware, but without attempting to illustrate all of the very extensive capabilities of ICE-85. As before, the paragraph numbers are keyed to the bold face numbers on the terminal session printout on facing pages.

1. Prior to the operation shown here, I execute the ISIS command ICE85. ICE-85 responds with a prompting asterisk. I execute the ICE-85 command

   LIST :F1:ONESEC.ICE

   which causes all subsequent terminal material, whether typed by me or produced by ICE-85, to be sent to the listing file named. I later print this file.

   I use the MAP command to specify that the block starting at the logical address 2000 (hex assumed by default) is to be placed in Intellec memory starting at 7000. Since I don't specify otherwise, this is a 2K block. I also map a block of specified length, F000 to FFFF, into the same addresses in Intellec memory, since my program uses the monitor, and likewise map the block starting at zero. ICE-85 warns me that I am mapping over the system, but since it is precisely the system I am trying to get at anyway, I'm not worried. The I/O ports that are used by the CO routine which my program calls are mapped to the Intellec; I have no option to map these into any other port numbers.

   Note that anything following a semicolon is treated as a comment, just as with the assembler.

2. I use the LOAD command to bring my program (ONESEC) in from the diskette on drive 1.

3. I use the SYMBOLS command to see the symbols (and their corresponding values, memory addresses in this case) that were brought in with the program when it was loaded. These symbols are available because I used the DEBUG option when I assembled the program. The line MODULE ..MODULE means that the name of the object program module is "MODULE"; this was assigned as a default name because I didn't specify otherwise in the LOCATE operation.

4. I ask to see the value of the program counter (PC). It should have been loaded with the value of the symbol START, because I put that symbol on the END operation in my program. It has indeed been loaded as desired.

5. I say GO, which is the simplest possible emulation command. My program runs as expected, at full speed, producing X's on the CRT (which is the console output device on the Microcomputer Development System I am using). Since this is program output rather than ICE-85 dialog, it does not show on the listing.

6. I press the ESC (escape) key on the console. ICE-85 resonds with the address (in the PC) of the next instruction that would have been executed if I had not interrupted the program.

7. I ask to see the registers as they stand at this point. The program counter is as noted; the stack pointer is two less than its starting value, which is right since there had been one PUSH in connection with the CALL; the A register has been decremented from its starting value; the flags I don't care about; the B register contains the 12 (decimal) placed in it at the beginning of the DELAY subroutine; the E, H, and L registers have not been used; the interrupt register I don't care about.

8. I define a new symbol, L2A, to be an address in the program at which I would like to terminate a later emulation. Note that a period must appear before every symbol, and that expressions may be used in defining symbols.

9. Now when I ask to see my symbols the new one is there. It is shown at the beginning of the listing to indicate that it is not among the symbols brought in with the module.

10. I ask for a display of the bytes beginning with the byte at the symbol just defined, with a length of two bytes. The purpose is to be sure that I defined the symbol properly, and that it does point to the instruction I intended. All is in order.

```
1  *MAP 2000 = INTELLEC 7000  ; LOWEST INTELLEC ADDRESS ALLOWED
   *MAP F000 TO FFFF = INTELLEC F000 ; MONITOR
   *MAP 0 = INTELLEC 0 ; MONITOR NEEDS THIS
   WARN C1:MAPPING OVER SYSTEM
   *MAP IO F0 TO FF = INTELLEC
2  *LOAD :F1:ONESEC
3  *SYMBOLS
   MODULE ..MODULE
   .CO=F809H
   .DELAY=2016H
   .L1=2003H
   .L2=2005H
   .L3=2018H
   .L4=2019H
   .START=2000H
4  *PC  ; LOADED FROM FILE?
   2000H
5  *GO
   EMULATION BEGUN
6  EMULATION TERMINATED, PC=2019H
   PROCESSING ABORTED
7  *REGISTERS
   P=2019H S=202EH A=07H F=14H B=0CH C=0AH D=27H E=00H H=00H L=00H I=00H
8  *DEFINE .L2A = .DELAY - 8
   *; THAT'S A LABEL ON "MVI C,'X'"
9  *SYMBOLS
   .L2A=200EH
   MODULE ..MODULE
   .CO=F809H
   .DELAY=2016H
   .L1=2003H
   .L2=2005H
   .L3=2018H
   .L4=2019H
   .START=2000H
10 *BYTE .L2A LENGTH 2 ; DISPLAY INSTRUCTION AT THAT ADDRESS
   200EH=0EH 58H
```

11. Now I begin emulation again, telling ICE-85 to begin execution with the instruction at START rather than picking up where the program was interrupted earlier, and to stop when the instruction beginning at L2A has been executed.

12. When emulation is terminated, the program counter (PC) is pointing at the instruction after the one at L2A.

13. The command RC displays the contents of the C register, which should just have been loaded with the letter X. ICE-85 responds with the hex value.

14. I could look this up, but instead I change the base to ASCII, so that the character will be printed out in its external form.

15. Now when I ask for the contents of the C register, I get the letter X as such.

16. I restore the base to hex.

17. I ask for the 20 instructions most recently executed to be printed. (I could also have asked for the last 20 machine cycles, or machine states; instructions is the default). ICE-85 responds with the instructions, oldest listed first, in "disassembled" form, that is, with mnemonic operation codes and register names rather than just hex bytes. The numbers at the extreme left are the trace buffer addresses, which don't concern us. The locations of the instructions and their mnemonic operation codes are shown. When a jump condition was satisfied the jump address appears, and otherwise not. For any instruction having a memory reference, the address and contents are shown, together with an indication whether the memory reference was a read (R), input (I), output (O), or write (W). The only memory references in this trace are to the stack, in the RET at 2021; we see that the return address (200A) was retrieved from the stack (202E and 202F).

18. I ask to begin again at the beginning, and stop after the instruction at DELAY is executed.

19. Emulation stops as requested.

20. I ask to see the stack pointer and the word at the top of the stack. That word is pointing to the instruction after the CALL of DELAY, as it should.

21. To prepare for what I want to do next, I set the contents of D register to 50 decimal (T for base ten).

22. I enable the DUMP operation, specifying that I want to dump the state of the machine after each CALL and RETURN.

23. In order to permit the testing and displaying that are required in the DUMP operation, the program will have to be able to run one STEP at a time. Such execution is much slower than real time. To speed up the program, I change the parameter in the DELAY call. Note the use of an expression in naming a byte to be changed, with enclosing parentheses.

24. For every CALL and RETURN executed until the contents of the D register have been reduced below 47, DUMP gives me the information shown. Note that the contents of the D register start at 32H = 50 decimal, and work down to 2FH = 47 decimal, as desired.

```
11 *GO FROM .START TILL .L2A EXECUTED
   EMULATION BEGUN
12 EMULATION TERMINATED, PC=2010H
13 *RC ; SHOULD HAVE JUST LOADED 'X'
   58H
14 *BASE = ASCII ; LET'S SEE IT IN ASCII
15 *RC
   X
16 *BASE = H
17 *PRINT -20 ; LOOK AT TRACE
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   0947: 2019 DCR C
   0949: 201A JNZ  2019
   0955: 2019 DCR C
   0957: 201A JNZ  2019
   0963: 2019 DCR C
   0965: 201A JNZ  2019
   0971: 2019 DCR C
   0973: 201A JNZ  2019
   0979: 2019 DCR C
   0981: 201A JNZ  2019
   0987: 2019 DCR C
   0989: 201A JNZ  2019
   0995: 2019 DCR C
   0997: 201A JNZ
   1001: 201D DCR A
   1003: 201E JNZ
   1007: 2021 RET          202E-R-0A 202F-R-20
   1013: 200A DCR D
   1015: 200B JNZ
   1019: 200E MVI C, 58
18 *GO FROM .START TILL .DELAY EXECUTED
   EMULATION BEGUN
19 EMULATION TERMINATED, PC=2018H
20 *WORD SP  ; WORD AT TOP OF STACK
   202EH=200AH
   *; 200AH IS RETURN ADDRESS OF 'DELAY'
21 *RD = 50T
22 *ENABLE DUMP CALL RETURN
23 *BYTE (.L2+1) = 2 ; CHANGE DELAY PARAMETER
24 *STEP FROM .START TILL RD < 47T
   EMULATION BEGUN
    2007-E-CD 2008-R-16 2009-R-20 202F-W-20 202E-W-0A
   P=2016H S=202EH A=02H F=54H B=0CH C=58H D=32H E=00H H=00H L=00H I=00H
    2021-E-C9 202E-R-0A 202F-R-20
   P=200AH S=2030H A=00H F=54H B=0CH C=00H D=32H E=00H H=00H L=00H I=00H
    2007-E-CD 2008-R-16 2009-R-20 202F-W-20 202E-W-0A
   P=2016H S=202EH A=02H F=10H B=0CH C=00H D=31H E=00H H=00H L=00H I=00H
    2021-E-C9 202E-R-0A 202F-R-20
   P=200AH S=2030H A=00H F=54H B=0CH C=00H D=31H E=00H H=00H L=00H I=00H
    2007-E-CD 2008-R-16 2009-R-20 202F-W-20 202E-W-0A
   P=2016H S=202EH A=02H F=14H B=0CH C=00H D=30H E=00H H=00H L=00H I=00H
    2021-E-C9 202E-R-0A 202F-R-20
   P=200AH S=2030H A=00H F=54H B=0CH C=00H D=30H E=00H H=00H L=00H I=00H
    2007-E-CD 2008-R-16 2009-R-20 202F-W-20 202E-W-0A
   P=2016H S=202EH A=02H F=00H B=0CH C=00H D=2FH E=00H H=00H L=00H I=00H
    2021-E-C9 202E-R-0A 202F-R-20
   P=200AH S=2030H A=00H F=54H B=0CH C=00H D=2FH E=00H H=00H L=00H I=00H
   EMULATION TERMINATED, PC=200BH
```

25. Now I want to transfer my program to prototype memory. I decide to check the memory mapping. All blocks are *guarded* (not defined) except the four blocks I mapped to Intellec memory.

26. I plug the ICE-85 umbilical cable into an SDK-85 board, which contains enough memory for my program. This can be thought of as a prototype system. Connecting the cable requires resetting the hardware, which means to let ICE-85 re-initialize its description of the hardware configuration.

27. I map the 2K block at 2000 to user memory.

28. I transfer the program from Intellec memory (actual physical location) to my "user memory" on the SDK-85 board, in actual locations 2000 and following, which is where this version of the program was LOCATEd to run.
29. The memory mapping now shows that the block at 2000 is user memory. The memory corresponding to the Intellec monitor is still being borrowed from the Intellec memory, and the input and output facilities are still borrowed.

30. Not quite convinced, I ask to see the contents of the memory locations where my program should be. It's there.

31. I start the program, which is now executing from a combination of user and Intellec memory.

32. The program stops under the same condition as given in step 24, which ICE-85 remembers, to save me the trouble of repeating the condition if I want it to apply again.

33. But I don't, so I give the GO command with a FOREVER to wipe out the previous stopping condition.

34. The program works just fine, except that the X's are sent out much too rapidly. *I use the ESC key to abort emulation.*

35. The problem is the DELAY parameter that I changed in step 23. I put it back to its original value.

36. Starting at the beginning now gives the expected behavior of one X per second.

37. I'm done, so I interrupt the program.

38. And exit from ICE-85. The next thing printed, which is not shown here because it was not done by ICE-85, was the ISIS message and a hyphen prompt.

    I now want to debug a program named ONEERR, which is related to the previous but contains a coding error. The intent is to send the characters A through Z to the system console, once each second, instead of just the letter X. (The program makes no provision for stopping after Z.) The erroneous program is shown in Figure 6-3.

39. I do the necessary memory mapping, taking into account that this program was LOCATEd to run in 4000.

40. I load the program.

41. I ask for the program symbols.

42. I start the program.

```
25 *MAP
   SHARED
   0000=I 0000     0800=G          1000=G          1800=G
   2000=I 7000     2800=G          3000=G          3800=G
   4000=G          4800=G          5000=G          5800=G
   6000=G          6800=G          7000=G          7800=G
   8000=G          8800=G          9000=G          9800=G
   A000=G          A800=G          B000=G          B800=G
   C000=G          C800=G          D000=G          D800=G
   E000=G          E800=G          F000=I F000     F800=I F800
   *
26 *RESET HARDWARE ; WE CHANGED ICE HARDWARE CONFIGURATION, NOW USING SDK-85
27 *MAP 2000 = USER
28 *BYTE 2000 = IBYTE 7000 TO 7023 ; MOVE PROGRAM FROM INTELLEC TO SDK MEMORY
29 *MAP
   SHARED
   0000=I 0000     0800=G          1000=G          1800=G
   2000=U          2800=G          3000=G          3800=G
   4000=G          4800=G          5000=G          5800=G
   6000=G          6800=G          7000=G          7800=G
   8000=G          8800=G          9000=G          9800=G
   A000=G          A800=G          B000=G          B800=G
   C000=G          C800=G          D000=G          D800=G
   E000=G          E800=G          F000=I F000     F800=I F800
   *;STILL BORROWING INTELLEC MEMORY FOR MONITOR
30 *BYTE 2000 TO 2023 ; VERIFY MOVE WORKED
   2000H=31H 30H 20H 16H 32H 3EH 02H CDH 16H 20H 15H C2H 05H 20H 0EH 58H
   2010H=CDH 09H F8H C3H 03H 20H 06H 0CH 48H 0DH C2H 19H 20H 3DH C2H 18H
   2020H=20H C9H 0AH 72H
   *GO FROM .START
   EMULATION BEGUN
31 EMULATION TERMINATED, PC=0024H
   *GO FROM .START FOREVER
   EMULATION BEGUN
32 EMULATION TERMINATED, PC=2018H
33 *GO FROM .START FOREVER
   EMULATION BEGUN
34 EMULATION TERMINATED, PC=2019H
   PROCESSING ABORTED
35 *BYTE (.L2+1) = 200T ;RESTORE DELAY PARAMETER
36 *GO FROM .START
   EMULATION BEGUN
37 EMULATION TERMINATED, PC=2019H
   PROCESSING ABORTED
38 *EXIT
39 *MAP 4000 = INTELLEC 7000
   *MAP F000 TO FFFF = INTELLEC F000
   *MAP 0 = INTELLEC 0
   WARN C1:MAPPING OVER SYSTEM
   *MAP IO F0 TO FF = INTELLEC
40 *LOAD :F1:ONEERR
41 *SYMBOLS
   MODULE ..MODULE
   .CO=F809H
   .CHAR=4028H
   .DELAY=401CH
   .L1=4006H
   .L2=4008H
   .L3=401EH
   .L4=401FH
   .START=4000H
42 *GO
   EMULATION BEGUN
```

43. The program works, after a fashion: the letter A is followed by random characters having no obvious relation to one another.

44. Detective work is required. I decide to stop emulation at the nearest convenient point after the loading of the character into the C register on line 18; execution of CO will do.

45. Execution terminates with the program counter in high memory, which is the monitor where CO is located.

46. I ask to see the three most recently executed instructions. The interesting one is the MOV, which read (R) from 4028 and obtained 41 hex, which is an A. All seems to be in order.

47. I say GO, without naming a break condition; the previous one (TILL .CO EXECUTED) is still in effect.

48. The MOV instruction now is highly suspicious: it obtains a character from 4029, whereas my intention in writing the program was that it should obtain a modified character from 4028. What is going on?

49. Try again.

50. The MOV address is still being modified.

51. If the program is really doing this, the H and L registers should contain 402B the next time around this loop. Let's STEP through the program until the combined registers contain this address.

52. There it is! Time for a hard look at the program. Sure enough, I've written it to modify the *contents* of the HL register instead of the *contents of the address pointed to* by the HL register. The old programming 101 confusion between address and contents!

53. To see if this really is the trouble, I program a four-byte fix in hex and enter the change in absolute. Naturally, I'll later go back and correct the source program and reassemble.

54. I start the corrected program.

55. Everything works as intended. I interrupt emulation.

56. And exit from ICE-85.

```
43 EMULATION TERMINATED, PC=4020H
   PROCESSING ABORTED
   *; CHARACTERS CAME OUT, BUT ALL WRONG
44 *GO FROM .START TILL .CO EXECUTED
   EMULATION BEGUN
45 EMULATION TERMINATED, PC=FD1DH
46 *PRINT -3
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   1003: 4011 MOV C,M      4028-R-41
   1007: 4012 CALL  F809   4036-W-40 4035-W-15
   1017: F809 JMP  FD1D
47 *GO ; SAME BREAK CONDITION AS BEFORE
   EMULATION BEGUN
   EMULATION TERMINATED, PC=FD1DH
48 *PRINT -3
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   1003: 4011 MOV C,M      4029-R-00
   1007: 4012 CALL  F809   4036-W-40 4035-W-15
   1017: F809 JMP  FD1D
49 *GO ; SAME BREAK CONDITION AS BEFORE
   EMULATION BEGUN
   EMULATION TERMINATED, PC=FD1DH
50 *PRINT -3
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   1003: 4011 MOV C,M      4029-R-00
   1007: 4012 CALL  F809   4036-W-40 4035-W-15
   1017: F809 JMP  FD1D
   *GO
   EMULATION BEGUN
   EMULATION TERMINATED, PC=FD1DH
   *PRINT -3
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   1003: 4011 MOV C,M      402A-R-00
   1007: 4012 CALL  F809   4036-W-40 4035-W-15
   1017: F809 JMP  FD1D
51 *STEP TILL RHL = 402B
   EMULATION BEGUN
   EMULATION TERMINATED, PC=4019H
52 *PRINT -5
         ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA ADDR-S-DA
   0999: FD43 MOV A,C
   1001: FD44 OUT  F6      F6F6-O-00
   1007: FD46 RET          4035-R-15 4036-R-40
   1013: 4015 LXI B, 0001
   1019: 4018 DAD B
53 *BYTE 4015 = C6,01,77,00  ; PATCH THE FIX
54 *GO FROM .START FOREVER
   EMULATION BEGUN
55 EMULATION TERMINATED, PC=401FH
   PROCESSING ABORTED
56 *EXIT
```

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0         MODULE   PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1          ; AN ASSEMBLY LANGUAGE PROGRAM TO SEND THE LETTER 'X'
                      2          ; TO THE CONSOLE OUTPUT DEVICE, ONCE EACH SECOND
                      3          ; A REVISED VERSION, INTENDED TO SEND THE CHARACTERS A THRU Z
                      4          ; TO THE CONSOLE, ONE PER SECOND
                      5          ; THIS PROGRAM CONTAINS DELIBERATE ERRORS!
                      6          ;
  F809                7 CO       EQU     0F809H  ; PROVIDE ADDRESS OF CO ROUTINE IN MONITOR
                      8          CSEG            ; MAKE THE SEGMENT RELOCATABLE
                      9          STKLN   2       ; SET STACK LENGTH
  0000 310000    S    10 START:  LXI     SP,STACK ; INITIALIZE STACK POINTER
  0003 212800    C    11         LXI     H,CHAR  ; LOAD H AND L WITH ADDRESS OF 'A'
  0006 1632           12 L1:     MVI     D,50    ; WILL CALL DELAY SUBROUTINE 50 TIMES
  0008 3EC8           13 L2:     MVI     A,200   ; SUBROUTINE PARAMETER
  000A CD1C00    C    14         CALL    DELAY   ; INVOKE SUBROUTINE
  000D 15             15         DCR     D
  000E C20800    C    16         JNZ     L2      ; 50 TIMES AROUND THIS LOOP = 1 SEC
                      17         ;
  0011 4E             18         MOV     C,M     ; MOVE CURRENT CHARACTER TO C REG
  0012 CD09F8         19         CALL    CO      ; MONITOR CONSOLE OUTPUT ROUTINE
  0015 010100         20         LXI     B,1     ; GET A CONSTANT 1 INTO B AND C
  0018 09             21         DAD     B       ; ADD TO H AND L
  0019 C30600    C    22         JMP     L1      ; AROUND THE LOOP INDEFINITELY
                      23         ;
                      24         ; THE DELAY SUBROUTINE
                      25         ;
  001C 060C           26 DELAY:  MVI     B,12
  001E 48             27 L3:     MOV     C,B
  001F 0D             28 L4:     DCR     C
  0020 C21F00    C    29         JNZ     L4
  0023 3D             30         DCR     A
  0024 C21E00    C    31         JNZ     L3
  0027 C9             32         RET
                      33         ;
  0028 41             34 CHAR:   DB      'A'     ; THE CHARACTER SENT TO CONSOLE - INITIALLY A
                      35         ;
  0000           C    36         END     START   ; PROGRAM EXECUTION BEGINS WITH SYMBOL 'START'

PUBLIC SYMBOLS


EXTERNAL SYMBOLS


USER SYMBOLS
CHAR   C 0028    CO     A F809    DELAY  C 001C    L1     C 0006    L2     C 0008    L3     C 001E    L4     C 001F
START  C 0000

ASSEMBLY COMPLETE,  NO ERRORS
```

FIGURE 6-3

Chapter 7

AN APPLICATION ILLUSTRATION

We shall now take a small, but representative, application and carry it through all of the steps of development to show how the pieces of the Intellec system and ISIS come into play. In the process of doing so, we shall provide a rudimentary roadmap of the operations carried out in getting a simple program running. The reader may find it useful to actually run the program — following this roadmap — before trying a program of his own.

The application we shall use is a simple traffic light controller. We imagine an intersection of a main street and a side street. The desired operation is that the light should stay green on the main street until a decision rule involving the arrival of cars on the side street and the amount of time they have been waiting has been satisfied. We suppose that there is a sensor in the pavement on the side street that sends an interrupt to the computer when a car arrives. We shall not include the control of a yellow caution light on either street; addition of this feature is one of several "exercises" that will be suggested at the end of the chapter.

Associated with each street is a time called the cycle length. In the program the variable named SIDE$CYCLE$LENGTH controls the fixed length of time that the light is green on the side street when that cycle is called into action. Even though the light stays green on the main street until the decision rule is satisfied, we need a variable named MAIN$CYCLE$LENGTH that is involved in the decision rule.

The decision rule is as follows. The side street gets a green cycle if either of the following conditions is satisfied:

1. Two or more cars are waiting on the side street and the main street has been green for a period of time greater than or equal to MAIN$CYCLE$LENGTH.
2. One car is waiting on the side street and the main street has been green for a period of time greater than or equal to two times MAIN$CYCLE$LENGTH.

THE SYSTEM ORGANIZATION

The system has one input and one output. The input is a signal that a car has arrived since the last time we sampled the input. The output goes to the traffic light controller; we assume that sending it a 1 makes the light on the main street green and the light on the side street red, and that sending it zero makes the light on the main street red and the light on the side street green.

A FIRST CUT AT THE PROGRAM DESIGN

Our first step in writing a program for this application is to think about the overall sequence of operations required, without at this stage getting enmeshed in details. One way to approach this is with *pseudocode*, as shown in Fig. 7-1. We see that the essential operations are described at a rather abstract level with the details left for later. This is

```
DO FOREVER;
   Display time since last change
   Add 1 to MAIN$TIME
   IF decision rule satisfied THEN
   DO;
      Cycle lights
      Set count of cars waiting to zero
      Set MAIN$TIME to zero
   END;
END;
```

FIGURE 7-1

clearly not a program, but it lets us organize our thinking about the required sequence of operations and lets us begin considering how to organize the parts of the program.

What has been shown here are the essentials of what is called the *main program loop*, which is the part of the program that repeats indefinitely as long as the system is in operation. This description of the main program assumes the existence of an interrupt procedure that is called into action when the arrival of a car on the side street causes an external interrupt of whatever is happening at that time in the program. When the interrupt procedure is invoked, all it has to do is add one to the count of cars waiting on the side street.

For the main program to do its job it will need to call upon several *procedures* — bodies of code that carry out specific functions. In the program that follows we shall use procedures from three different sources:

1. Some procedures will be programmed and compiled as part of our PL/M program.

2. One procedure will be coded in assembly language, assembled in a separate operation, and then linked with the compiled PL/M program.

3. One procedure that is needed only for development purposes and which would not be part of an operational program will be "borrowed" from the monitor.

This modular organization of a program into procedures is recommended; it saves the time and trouble of writing any procedures that are already available, tends to simplify program checkout, and facilitates breaking large programs into manageable pieces that can be programmed in parallel by several programmers. PL/M procedures can be compiled separately and then LINKed together when needed, or LINKed with Fortran or assembly language modules.

THE TRAFFIC LIGHT CONTROLLER PROGRAM

The PL/M portion of the program for this application is shown in Fig. 7-2. We can study it by taking a look first at the declarations, then noting the main program, then examining the procedures in the sequence in which they appear.

The declarations at the beginning contain two features of some interest. Statement 6 contains a PL/M *macro*, which is somewhat similar in concept but quite different in detail from an assembler macro. The declaration means that anywhere the identifier FOREVER appears in our program, it will be replaced — before compilation — by WHILE 1. The program listing will continue to show the FOREVER, aiding understandability, but what the compiler sees will be the syntactically correct form. Much more elaborate constructions are possible.

Statements 7-9 are messages that will be sent to the CRT for developmental purposes, during program checkout. Since we really don't need to know how many characters there are in these messages, we use the asterisk notation in declaring them to be arrays of bytes, and let the compiler count them in order to allocate memory.

The main program begins, in statements 50-54, with some operations required to initialize the interrupt logic in the Intellec Microcomputer Development System. Any production system would probably require some such operations, but the details are not of interest to us here.

The remaining part of the main program, statements 55-67, is very similar to the pseudocode already displayed.

The first procedure, in statements 10-13, is entered automatically when — and only when — the external interrupt number 4 is detected. For developmental purposes we shall use the interrupt key on the front panel of the Intellec chassis; in the final product the interrupt would of course come from the street sensor. When the interrupt is sensed by the Intellec circuitry the instruction then being executed is completed and control of the microprocessor is turned over to the interrupt procedure. The interrupt procedure saves all register contents. When the interrupt procedure has finished its work the registers are restored and the program that was being executed when the interrupt arrived is resumed. There is considerably more to the full story, but that's the basic idea.

It will be convenient in this program to have a procedure that delays for 0.01 second times the value of the argument, rather than the 100 μs of the PL/M procedure named TIME that we saw in Chapter 4. This is best coded in assembly language, and later combined with the compiled PL/M program. This program, which is shown in Fig. 7-3, is called DELAY. We see that it contains a notification to the assembler that the name DELAY is "public," that is, it needs to be known outside of this program. Correspondingly, in the PL/M program, in statement 13 we tell the compiler that DELAY is "external" to that program, that is, defined elsewhere. The LINK program will later use this information to pull the pieces together. As we write these programs we have no idea where they will eventually reside in the product's memory, which is a significant advantage.

The procedure named CO defined in statements 17-19 as being external to the PL/M program, is in fact permanently available in the monitor, at the high end of the Intellec memory. We will later use the LINK operation to tell the compiled PL/M program where it is.

The procedure named DISPLAY, defined in statements 20-39, presents information about the traffic light process that will be needed to check out the program. Notice that it has its own declarations, which are accordingly *local* to this procedure: the variables declared here are not "known" outside of this procedure. On the other hand, the variables declared at the beginning of the entire program are *global* to this procedure and *are* known within the procedure. The local/global mechanism can be used to good advantage to limit unwanted interactions between procedures written by several programmers.

The iterative DO statements (24, 27, and 34) that send the characters of the three messages to the CRT use the built-in procedure LAST, which supplies the element number of the last character of the named array. This built-in procedure is in a library named PLM80.LIB that is provided with the compiler, and which is brought in either automatically or at the LINK stage, depending on details that don't interest us.

The CYCLE procedure, in statements 40-49, runs the traffic light through its side-green cycle for the length of time specified by the value of the variable named SIDE$CYCLE$LENGTH.

```
                /* TRAFFIC LIGHT CONTROLLER PROGRAM */

  1             CARS:
                DO;
  2    1           DECLARE (MAIN$TIME, SIDE$TIME) BYTE;
  3    1           DECLARE MAIN$CYCLE$LENGTH BYTE DATA(8), SIDE$CYCLE$LENGTH BYTE DATA(5);
  4    1           DECLARE CARS$WAITING BYTE;
  5    1           DECLARE LIGHT$STATUS BYTE;
  6    1           DECLARE FOREVER LITERALLY 'WHILE 1';
  7    1           DECLARE MAIN$GREEN$MESSAGE(*) BYTE DATA('MAIN GREEN, SIDE RED');
  8    1           DECLARE SIDE$GREEN$MESSAGE(*) BYTE DATA('SIDE GREEN, MAIN RED');
  9    1           DECLARE TIME$MESSAGE(*) BYTE DATA('  SECS SINCE LIGHT CHANGE');


                   /* FOLLOWING PROCEDURE ENTERED ONLY BY INTERRUPT */
 10    1           SIDE$STREET$CAR:
                   PROCEDURE INTERRUPT 4;
 11    2              CARS$WAITING = CARS$WAITING + 1;
 12    2              OUTPUT(0FDH) = 20H;      /* RESTORE INTELLEC INTERRUPT LOGIC */
 13    2           END SIDE$STREET$CAR;

                   /* FOLLOWING PROCEDURE CODED IN ASSEMBLY LANGUAGE AND LINKED IN */
 14    1           DELAY:
                   PROCEDURE(TIME$HUNDREDTHS) EXTERNAL;
 15    2              DECLARE TIME$HUNDREDTHS BYTE;
 16    2           END DELAY;

                   /* FOLLOWING PROCEDURE BORROWED FROM THE MONITOR */
 17    1           CO:
                   PROCEDURE(CHAR) EXTERNAL;
 18    2              DECLARE CHAR BYTE;
 19    2           END CO;

 20    1           DISPLAY:
                   PROCEDURE (CYCLE$TIME);
 21    2              DECLARE CYCLE$TIME BYTE;
 22    2              DECLARE I BYTE;

 23    2              IF LIGHT$STATUS = 0 THEN
 24    2              DO I = 0 TO LAST(SIDE$GREEN$MESSAGE);
 25    3                 CALL CO(SIDE$GREEN$MESSAGE(I));
 26    3              END;
                      ELSE
 27    2              DO I = 0 TO LAST(MAIN$GREEN$MESSAGE);
 28    3                 CALL CO(MAIN$GREEN$MESSAGE(I));
 29    3              END;
 30    2              CALL CO(0DH);     /* CARRIAGE RETURN */
 31    2              CALL CO(0AH);     /* LINE FEED */
 32    2              CALL CO( (CYCLE$TIME / 10) OR 30H );  /* TEN'S DIGIT */
 33    2              CALL CO( (CYCLE$TIME MOD 10) OR 30H); /* UNIT'S DIGIT */
 34    2              DO I = 0 TO LAST(TIME$MESSAGE);

 35    3                 CALL CO(TIME$MESSAGE(I));
 36    3              END;
 37    2              CALL CO(0DH);     /* CARRIAGE RETURN */
 38    2              CALL CO(0AH);     /* LINE FEED */
 39    2           END DISPLAY;
```

FIGURE 7-2 (Sheet 1 of 2)

```
40   1          CYCLE:
                PROCEDURE;
41   2             LIGHT$STATUS = 0;  /* MAIN RED, SIDE GREEN */
42   2             SIDE$TIME = 0;
43   2             DO WHILE SIDE$TIME <= SIDE$CYCLE$LENGTH;
44   3                CALL DISPLAY(SIDE$TIME);
45   3                CALL DELAY(100);
46   3                SIDE$TIME = SIDE$TIME + 1;
47   3             END;
48   2             LIGHT$STATUS = 1;   /* MAIN GREEN, SIDE RED */
49   2          END CYCLE;

               /* MAIN PROGRAM -- EXECUTION BEGINS HERE */

                /* PREPARE INTELLEC INTERRUPT LOGIC */
50   1          DISABLE;
51   1          OUTPUT(0FDH) = 12H;     /* INITIALIZE INTELLEC INTERRUPT LOGIC */
52   1          OUTPUT(0FCH) = 0;       /* DITTO */
53   1          OUTPUT(0FCH) = 0E0H;    /* ACCEPT INTERRUPTS 0-4 */
54   1          ENABLE;

55   1          CARS$WAITING = 0;
56   1          MAIN$TIME = 0;
57   1          DO FOREVER;
58   2             CALL DISPLAY(MAIN$TIME);
59   2             CALL DELAY(100);
60   2             MAIN$TIME = MAIN$TIME + 1;
61   2             IF    (CARS$WAITING >= 2) AND (MAIN$TIME >= MAIN$CYCLE$LENGTH)
                      OR (CARS$WAITING = 1) AND (MAIN$TIME >= 2 * MAIN$CYCLE$LENGTH) THEN
62   2             DO;
63   3                CALL CYCLE;
64   3                CARS$WAITING = 0;
65   3                MAIN$TIME = 0;
66   3             END;
67   2          END;

68   1       END CARS;
```

FIGURE 7-2 (Sheet 2 of 2)

```
ASM80 :F1:DELAY.SRC

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0         MODULE   PAGE    1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1          ; TIME DELAY SUBROUTINE - DELAYS 0.01 SEC TIMES NUMBER IN A REG
                      2          ;
                      3          CSEG
                      4          PUBLIC  DELAY
  0000 79             5 DELAY:   MOV     A,C     ; PL/M LINKAGE CONVENTION PUTS ARGUMENT IN C REG
  0001 06FF           6          MVI     B,255   ; NOTE CHANGED PARAMETER TO GET LONGER DELAY
  0003 48             7 LAB1:    MOV     C,B
  0004 0D             8 LAB2:    DCR     C
  0005 221600    C    9          SHLD    TEMP    ; WASTE 14 CYCLES
  0008 221600    C   10          SHLD    TEMP    ; DITTO
  000B 221600    C   11          SHLD    TEMP    ; DITTO
  000E C20400    C   12          JNZ     LAB2
  0011 3D            13          DCR     A
  0012 C20300    C   14          JNZ     LAB1
  0015 C9            15          RET
                     16          ;
  0002               17 TEMP:    DS      2       ; PLACE TO STORE H AND L -- CONTENTS NEVER USED
                     18          END

PUBLIC SYMBOLS
DELAY  C 0000

EXTERNAL SYMBOLS


USER SYMBOLS
DELAY  C 0000    LAB1   C 0003    LAB2   C 0004    TEMP   C 0016

ASSEMBLY COMPLETE,  NO ERRORS
```

FIGURE 7-3

THE STEPS IN ENTERING AND RUNNING A PROGRAM

To help you through the steps of getting your first program running on the Microcomputer Development System we shall walk through the operations necessary to get this program running. The approach here will be in a somewhat cookbook fashion to help you get started. As you become familiar with the various ISIS operations, you will quickly branch out. Here then is what you need to do.

1. Turn on the Intellec components. Insert an ISIS system diskette in drive 0 and a blank diskette in drive 1.

2. Press the Reset key on the Intellec console and release it. After a brief interval the message

   ISIS-II, Vx.y

   will be produced at the console, where x.y will be numbers indicating the Version number of your ISIS system. (New versions of most programs are issued from time to time.) ISIS will then produce a dash, telling you it is ready to accept a command. Only ISIS prompts with a dash, so any time you see a dash prompt you know you are dealing with ISIS, not the monitor, text editor, ICE, or the Library Manager, which use different prompt characters.

3. Type in the command

   FORMAT MYDISK.DDM

   Actually you may use any combination of six or fewer characters before the dot and any combination of three or fewer after. What comes after the dot might be your initials or the date or anything else you please. (If you are working with a disk already formatted from use with previous chapters, of course skip this step.)

4. Type in the command

   EDIT :F1:CARS.SRC

   The name CARS is your choice, but the rest must be as shown. After identifying the text editor version and giving a notification that this is a new file, the system will respond with an asterisk prompt, which tells you that you are dealing with the text editor. You may now enter any text editor command that you wish. Since you are dealing with a new file, the only command that makes any sense is Insert. You accordingly type an I and then as much of the text of the program as you wish. When you wish to end the Insert command, hit the ESC or ALT MODE key twice, which will be echoed to you as two dollar signs. During this input you can use the rubout key to erase the immediately preceding character, which will be echoed back to you. You may use Control-R to get a new copy of the present line minus any deletions. You may use Control-X to delete the present line altogether and make a fresh try.

   Once you have escaped from the Insert command, whether or not you have typed in the entire program, you may use the various other text editor commands to alter the program as described in Chapter 3. It is not necesssry to learn everything about the text editor, but do familiarize yourself with Find and Search commands since they are so useful. Remember that the pointer always resides *between* two characters. If you want to insert a new line you must be pointing at the beginning of the line *after* the point where you want the insertion.

   If you have typed in only part of the program and want to enter some more of it, use the Z command to get to the end of the file before beginning with a new Insert.

   When you have finished entering the program, or if you need to stop for any reason, use the E to exit from the text editor, which automatically stores the file on diskette under the name given in the EDIT command.

   Any time you wish to modify this file, such as when compiler diagnostics point out typing errors to you or whatever, you once again type the command

   EDIT :F1:CARS.SRC

Now when the text editor comes into play it will not inform you this is a new file. In order to work on your file you must get it into the text editor's memory area using the Append command.

Recall that this command brings in a maximum of 50 lines. You can always see whether you have gotten in your entire file by using the sequence Z-LT to see whether the line produced is the last one in the file on diskette.

To study your program — if you have a line printer — you can get a hard copy of it by entering the ISIS command

```
COPY :F1:CARS.SRC TO :LP:
```

(You must Exit from the text editor to do this.) If you don't have a line printer you can send the program to the console device by typing in the ISIS command

```
COPY :F1:CARS.SRC TO :CO:
```

(Remember that if your output device is a CRT, this will be much too fast for you to read. You can stop the output with Control-S and start it again with Control-Q.)

5. When you are satisfied that your program is correct you can compile it with the ISIS command

```
PLM80 :F1:CARS.SRC. DEBUG
```

This will take a minute or two. At the end of the compilation you will get a message noting the completion and specifying how many errors were detected by the compiler.

Two new files will have been created by the compilation, having file names

```
:F1:CARS.OBJ and :F1:CARS.LST
```

OBJ stands for object; this file is the program ready to be combined with ISIS procedures to produce a program that can actually run. LST stands for list and is a file that contains the statement numbers, page numbers, etc., that have been shown in the PL/M programs in this manual. If you want a copy of the listing enter the command

```
COPY :F1:CARS.LST TO :LP:
```

If you don't have a line printer or if you simply want to see what the diagnostic error messages are, you can see the listing file on the system console by entering

```
COPY :F1:CARS.LST TO :CO:
```

6. If the compiler has detected errors you can use the diagnostic error messages in the listing to see what they are and determine what changes need to be made in the source program. Make these changes using the text editor. When you have edited the program, the operation of executing the text editor command E will save the previous version of that file on a file named

```
:F1:CARS.BAK
```

for safety.

Now you recompile, etc., eventually arriving at a compilation showing zero errors. This of course does not prove there aren't any logic errors or simple typographical errors that would not create any error conditions, but you will at least be ready to give the program a try.

7. Again using the text editor, create a file named :F1:DELAY.SRC containing the assembly language program shown earlier.

8. Assemble it using the command

   ASM80 :F1:DELAY.SRC DEBUG

9. Get a copy of the listing file with the command

   COPY :F1:DELAY.LST TO :LP:

10. If there are any errors, you can use the diagnostic messages in the listing to see what your typing errors were, correct them using the text editor, and reassemble. (An assembler error message appears as a single letter in the left margin of the listing.)

11. To combine your object programs with the ISIS routines that you have specified and certain other routines such as procedures for doing multiplication and division that are not present in the 8080 CPU, we now carry out the LINK operation.

    Type the command

    LINK :F1:CARS.OBJ, :F1:DELAY.OBJ, SYSTEM.LIB, PLM80.LIB TO :F1:CARS.LNK

    Actually you can type almost anything you please where we have shown LNK, except it must be different from OBJ.

12. The file named :F1:CARS.LNK now contains your program almost ready to run. The only remaining operation is to place the program and data in actual memory locations to convert them from the relative form in which they occur now. This assignment can be done with a great deal of flexibility as described in the manuals. For our purposes all we need to do is type the command

    LOCATE :F1:CARS.LNK

    The output of this operation is a file having the name :F1:CARS, with no extension following the file name.

13. You can now run your program simply by typing in

    :F1:CARS

    That name, all by itself, is now a command to execute the program. If everything has been done correctly, your program will now begin to run. In the case of the traffic light controller, it will give the status of the system every second. Push interrupt 4 on the Intellec console and see what happens.

AN ICE SESSION

We may now see how in-circuit emulationcould be used to check out this program. To make the process interesting, we shall insert some errors in the program and see how ICE-85 might help us find them. Figure 7-4 shows the program of Fig. 7-2, modified in two ways. First, it has two deliberate errors, which we shall find using ICE-85. Second, the statements used in connection with the Intellec interrupt logic have been removed; these make it impossible to use ICE-85, so we shall handle the interrupt in another way. (You can, of course, use ICE-85 with interrupts in your prototype.)

In particular, the procedure

SIDE$STREET$CAR

no longer has the INTERRUPT attribute, and the main program no longer has the steps to initialize the Intellec interrupt logic.

```
                /* TRAFFIC LIGHT CONTROLLER PROGRAM */
                /* THIS PROGRAM CONTAINS DELIBERATE ERRORS! */

 1              CARS:
                DO;
 2    1            DECLARE (MAIN$TIME, SIDE$TIME) BYTE;
 3    1            DECLARE MAIN$CYCLE$LENGTH BYTE DATA(8), SIDE$CYCLE$LENGTH BYTE DATA(5);
 4    1            DECLARE CARS$WAITING BYTE;
 5    1            DECLARE LIGHT$STATUS BYTE;
 6    1            DECLARE FOREVER LITERALLY 'WHILE 1';
 7    1            DECLARE MAIN$GREEN$MESSAGE(*) BYTE DATA('MAIN GREEN, SIDE RED');
 8    1            DECLARE SIDE$GREEN$MESSAGE(*) BYTE DATA('SIDE GREEN, MAIN RED');
 9    1            DECLARE TIME$MESSAGE(*) BYTE DATA('  SECS SINCE LIGHT CHANGE');


                   /* IN EARLIER VERSION, THIS PROCEDURE WAS ENTERED ONLY BY INTERRUPT */
                   /* NOW, IT IS ENTERED THROUGH AN ICE CALL */
                   /* ALSO, THE INTELLEC LOGIC RESTORATION HAS BEEN DELETED */
10    1            SIDE$STREET$CAR:
                   PROCEDURE;
11    2               SIDE$TIME = SIDE$TIME + 1;
12    2            END SIDE$STREET$CAR;

                   /* FOLLOWING PROCEDURE CODED IN ASSEMBLY LANGUAGE AND LINKED IN */
13    1            DELAY:
                   PROCEDURE(TIME$HUNDREDTHS) EXTERNAL;
14    2               DECLARE TIME$HUNDREDTHS BYTE;
15    2            END DELAY;

                   /* FOLLOWING PROCEDURE BORROWED FROM THE MONITOR */
16    1            CO:
                   PROCEDURE(CHAR) EXTERNAL;
17    2               DECLARE CHAR BYTE;
18    2            END CO;

19    1            DISPLAY:
                   PROCEDURE (CYCLE$TIME);
20    2               DECLARE CYCLE$TIME BYTE;
21    2               DECLARE I BYTE;

22    2               IF LIGHT$STATUS = 0 THEN
23    2               DO I = 0 TO LAST(SIDE$GREEN$MESSAGE);
24    3                  CALL CO(SIDE$GREEN$MESSAGE(I));
25    3               END;
                      ELSE
26    2               DO I = 0 TO LAST(MAIN$GREEN$MESSAGE);
27    3                  CALL CO(MAIN$GREEN$MESSAGE(I));
28    3               END;
29    2               CALL CO(0DH);     /* CARRIAGE RETURN */
30    2               CALL CO(0AH);     /* LINE FEED */
31    2               CALL CO( (CYCLE$TIME / 10) OR 30H );  /* TEN'S DIGIT */
```

FIGURE 7-4 (Sheet 1 of 2)

```
32   2            CALL CO( (CYCLE$TIME MOD 10) OR 30H); /* UNIT'S DIGIT */
33   2            DO I = 0 TO LAST(TIME$MESSAGE);
34   3               CALL CO(TIME$MESSAGE(I));
35   3            END;
36   2            CALL CO(0DH);     /* CARRIAGE RETURN */
37   2            CALL CO(0AH);     /* LINE FEED */
38   2         END DISPLAY;

39   1         CYCLE:
          PROCEDURE;
40   2            LIGHT$STATUS = 0;  /* MAIN RED, SIDE GREEN */
41   2            SIDE$TIME = 0;
42   2            DO WHILE SIDE$TIME <= SIDE$CYCLE$LENGTH;
43   3               CALL DISPLAY(SIDE$TIME);
44   3               CALL DELAY(100);
45   3               SIDE$TIME = SIDE$TIME + 1;
46   3            END;
47   2            LIGHT$STATUS = 1;    /* MAIN GREEN, SIDE RED */
48   2         END CYCLE;

              /* MAIN PROGRAM -- EXECUTION BEGINS HERE */
              /* INTELLEC LOCIC INITIALIZATION HAS BEEN DELETED */

49   1         CARS$WAITING = 0;
50   1         MAIN$TIME = 0;
51   1         LIGHT$STATUS = 1; /* MAIN GREEN INITIALLY */
52   1         DO FOREVER;
53   2            CALL DISPLAY(MAIN$TIME);
54   2            CALL DELAY(100);
55   2            MAIN$TIME = MAIN$TIME + 1;
56   2            IF    (CARS$WAITING >= 2) AND (MAIN$TIME >= MAIN$CYCLE$LENGTH)
                     AND (CARS$WAITING = 1) AND (MAIN$TIME >= 2 * MAIN$CYCLE$LENGTH) THEN
57   2            DO;
58   3               CALL CYCLE;
59   3               CARS$WAITING = 0;
60   3               MAIN$TIME = 0;
61   3            END;
62   2         END;

63   1      END CARS;
```

FIGURE 7-4 (Sheet 2 of 2)

1. I map memory and I/O as required for my program. Note the use of abbreviations: LEN for LENGTH, INT for INTELLEC. All ICE-85 words can be abbreviated to three letters, and to one or two in some cases.

2. I load my program.

3. PC (the program counter) has been loaded with the starting address of the program; I give a symbol named START this value so that I can later restart the program from the beginning.

4. We will be referring many times to the routine named SIDE$STREET$CAR, which is a lot of typing. I define a shorter symbol to have the same value.

5. I ask for the symbols, which are printed in two groups, one for the PL/M program and one for the assembly language subroutine.

6. I start emulation.

7. Emulation is stopped with the ESC key. The output to the CRT had been correct.

8. The ICE-85 command CALL brings the former interrupt routine into play, just as though it had been entered by an interrupt. In other words, after the procedure has been executed, control returns to whatever had been going on when I pressed the ESC key.

9. I interrupt emulation with the ESC key again.

10. A second CALL represents a second car.

11. The light did not cycle, even though enough time had passed for the decision rule to have been satisfied with two cars. What is happening?

12. What is the value of CARS$WAITING? It should be 2 after two calls of SIDE$STREET$CAR. But it's zero!

13. Well, let's see if the procedure CYCLE is ever entered.

14. It isn't.

15. Let's see if the IF statement embodying the decision rule is ever executed. This was statement 56 in the program.

16. Yes it is. That's not the trouble.

17. Curious, I ask again for the value of CARS$WAITING. It still isn't being incremented.

```
1 *MAP 3000 LEN 4K = INT 7000 ; PROGRAM LOCATED AT 3680H
  *MAP 0 = INT 0 ; MONITOR VARIABLES
  WARN C1:MAPPING OVER SYSTEM
  *MAP F800 = INT F800 ; MONITOR
  *MAP IO F0 TO FF = INTELLEC
2 *LOAD :F1:CARS5
3 *DEFINE .START = PC ; START ADDRESS
4 *DEFINE .I4 = .SIDE$STREET$CAR ; SIMULATED INTERRUPT 4 ROUTINE
5 *SYMBOLS
  .START=36C3H
  .I4=3729H
  MODULE ..CARS
  .MEMORY=3882H
  .MAINTIME=387CH
  .SIDETIME=387DH
  .MAINCYCLELENGTH=3680H
  .SIDECYCLELENGTH=3681H
  .CARSWAITING=387EH
  .LIGHTSTATUS=387FH
  .MAINGREENMESSAGE=3682H
  .SIDEGREENMESSAGE=3696H
  .TIMEMESSAGE=36AAH
  .SIDESTREETCAR=3729H
  .DISPLAY=372EH
  .CYCLETIME=3880H
  .I=3881H
  .CYCLE=37E2H
  MODULE ..MODULE
  .DELAY=380FH
  .LAB1=3812H
  .LAB2=3813H
  .TEMP=3829H
6 *GO
  EMULATION BEGUN
7 EMULATION TERMINATED, PC=3820H
  PROCESSING ABORTED
  ;MAIN GREEN
8 *CALL .I4 ; FIRST CAR
  EMULATION BEGUN
9 EMULATION TERMINATED, PC=381AH
  PROCESSING ABORTED
  ;MAIN GREEN
10 *CALL .I4 ; SECOND CAR
  EMULATION BEGUN
11 EMULATION TERMINATED, PC=381DH
  PROCESSING ABORTED
  *;STILL MAIN GREEN, EVEN AFTER 2 CARS
12 *BYTE .CARS$WAITING ; SHOULD BE 2
  387EH=00H
13 *GO TILL .CYCLE EXECUTED ; DO WE REACH CYCLE?
  EMULATION BEGUN
14 EMULATION TERMINATED, PC=3814H
  PROCESSING ABORTED
  *;NO.  STILL MAIN GREEN
15 *G T #56 E ; DO WE REACH DECISION POINT? (NOTE USE OF ABBREVIATIONS)
  EMULATION BEGUN
16 EMULATION TERMINATED, PC=36E8H
  *;YES.  WHY DID WE NOT REACH CYCLE?
17 *BYTE .CARS$WAITING ; SHOULD BE 2
  387EH=00H
```

18. Now I want to see if CARS$WAITING is ever written, which it should be in the "interrupt" procedure SIDE$STREET$CAR.

    Since this procedure will be entered with a CALL and there is no way to say "CALL TILL .CARS$WAITING WRITTEN," I set a breakpoint.

19. I CALL the procedure.

20. Emulation did not stop; CARS$WAITING is never being written.

21. Let's look at the code for the assignment statement. The expression #11 gives the address of the first byte of statement 11. The expression #12-1 gives the address of the byte before the start of statement 12, which is the last byte of the code generated for statement 11. The code is seen to consist of four bytes. The middle two bytes are an address, 387DH. This should be the address of CARS$WAITING.

22. Now we finally notice that the assignment statement 11 is wrong: it says to increment SIDE$TIME instead of CARS$WAITING. To confirm this, we ask to see the address of SIDE$TIME, which is indeed 387DH.

23. I change this word to the address of CARS$WAITING.

24. I check to be sure the change was made correctly. The address has been changed to 387EH.

25. This is indeed the address of CARS$WAITING.

26. Try it.

27. Interrupt with ESC.

28. First car.

29. Interrupt with ESC.

30. Second car.

31. Now what? It still doesn't cycle.

32. Well, has CARS$WAITING been incremented? Yes, it has.

33. I decide to give up for the day, but want to save the program as it now stands, i.e., with the patched change and with the symbols I defined.

34. I exit from ICE-85.

35. Coming back the next day, I must do the mapping again.

36. I load the program, as modified in yesterday's session.

```
18 *GR = TILL .CARS$WAITING WRITTEN ; SET UP BREAKPOINT
19 *CALL .I4
   EMULATION BEGUN
20 EMULATION TERMINATED, PC=3813H
   PROCESSING ABORTED
   *;IT DIDN'T GET WRITTEN.
21 *BYTE #11 TO (#12-1) ; EXAMINE CURRENT CODE
   3729H=21H 7DH 38H 34H
22 *.SIDE$TIME
   387DH
   *;CHANGE 'SIDE$TIME' TO 'CARS$WAITING'
23 *WORD (#11+1) = .CARS$WAITING
24 *BYTE #11 TO (#12-1) ; VERIFY
   3729H=21H 7EH 38H 34H
25 *.CARS$WAITING
   387EH
26 *GO FROM .START FOREVER ; TRY IT OUT
   EMULATION BEGUN
27 EMULATION TERMINATED, PC=381DH
   PROCESSING ABORTED
   *;MAIN GREEN
28 *CALL .I4 ; FIRST CAR
   EMULATION BEGUN
29 EMULATION TERMINATED, PC=3820H
   PROCESSING ABORTED
   *;MAIN GREEN
30 *CALL .I4 ; SECOND CAR
   EMULATION BEGUN
31 EMULATION TERMINATED, PC=381AH
   PROCESSING ABORTED
   *;STILL MAIN GREEN
32 *BYTE .CARS$WAITING ; SHOULD BE 2
   387EH=02H
   *;THE PATCH SEEMS CORRECT.  WHY DOESN'T IT CYCLE?
   *;LET'S SAVE WHAT WE'VE DONE AND CONTINUE DEBUGGING LATER
33 *SAVE :F1:CARS5.SAV
34 *EXIT
35 *MAP 3000 LENGTH 4K = INTELLEC 7000
   *MAP 0 = INT 0
   WARN C1:MAPPING OVER SYSTEM
   *MAP F800 = INT F800
   *MAP IO F0 TO FF = INT
36 *LOAD :F1:CARS5.SAV ; LOAD WHAT WE SAVED EARLIER
```

37. I am curious to see if the new symbols are there. They are. Note that I4 has the same value as SIDE$STREET$CAR.

38. I try an emulation.

39. And interrupt.

40. First car.

41. Interrupt.

42. Has CARS$WAITING been incremented? Yes, so yesterday's patch must be in the program.

43. Second car.

44. Interrupt. The light didn't cycle.

45. Is CARS$WAITING right? Yes.

46. Is the IF statement that implements the decision rule being entered?

47. Yes.

48. Do we get past that statement, and if so, to where?

49. Emulation stops — but on which following statement? PC contents shown give the address of the *next* instruction; I want to know what was *previously* executed.

50. PPC gives the *previous* program counter contents.

51. OK, what is the address of the first byte of code for statement 62? Since it is the same as the value of PPC, we must just have executed statement 62. Therefore we *are* getting past the IF statement.

52. So what is the value of CARS$WAITING? It's 2!

53. How about MAIN$TIME? 0BH is 11 decimal. With 2 cars waiting and 11 seconds having passed, the decision rule should have been satisfied.

    I have now narrowed the trouble area down to the IF statement. ICE-85 can do no more to tell me what is wrong, when a syntactically correct statement simply gives wrong results. With my attention forced to focus on this one statement, I eventually will see that the logic is wrong; the AND at the beginning of the second line of the statement should be an OR.

    The solution is to recompile the program, which must be done eventually anyway. Machine language patches have their limits, and this error would be rather hard to patch.

54. I exit from ICE-85, correct the program, and try it again. It works correctly.

```
37 *SYMBOLS
   MODULE ..CARS
   .START=36C3H
   .I4=3729H
   .MEMORY=3882H
   .MAINTIME=387CH
   .SIDETIME=387DH
   .MAINCYCLELENGTH=3680H
   .SIDECYCLELENGTH=3681H
   .CARSWAITING=387EH
   .LIGHTSTATUS=387FH
   .MAINGREENMESSAGE=3682H
   .SIDEGREENMESSAGE=3696H
   .TIMEMESSAGE=36AAH
   .SIDESTREETCAR=3729H
   .DISPLAY=372EH
   .CYCLETIME=3880H
   .I=3881H
   .CYCLE=37E2H
   MODULE ..MODULE
   .DELAY=380FH
   .LAB1=3812H
   .LAB2=3813H
   .TEMP=3829H
   *;NOTICE ALL SYMBOLS ARE THERE, INCLUDING THOSE DEFINED IN PREVIOUS SESSION
38 *GO
   EMULATION BEGUN
39 EMULATION TERMINATED, PC=3817H
   PROCESSING ABORTED
   *;MAIN GREEN
40 *CALL .I4
   EMULATION BEGUN
41 EMULATION TERMINATED, PC=3817H
   PROCESSING ABORTED
   *;MAIN GREEN
42 *BYTE .CARS$WAITING
   387EH=01H
   *;HENCE PATCH WORKED
43 *CALL .I4
   EMULATION BEGUN
44 EMULATION TERMINATED, PC=381AH
   PROCESSING ABORTED
   *;STILL MAIN GREEN
45 *BYTE .CARS$WAITING
   387EH=02H
46 *GO TILL #56 E ; RIGHT BEFORE DECISION POINT
   EMULATION BEGUN
47 EMULATION TERMINATED, PC=36E8H
48 *GO TILL #57 E OR #62 E
   EMULATION BEGUN
49 EMULATION TERMINATED, PC=36D5H
50 *PPC ; LAST INSTRUCTION EXECUTED
   3724H
51 *#62
   3724H
   *;SO WE BRANCHED AROUND, THE IF CONDITION WAS NOT SATISFIED
52 *BYTE .CARS$WAITING
   387EH=02H
53 *BYTE .MAIN$TIME
   387CH=0BH
   *;IT IS A LOGIC PROBLEM IN IF CONDITION.  MUST RECOMPILE
54 *EXIT
```

THINGS TO TRY

If you want to use this program for some practice in using the Intellec system, here are some modifications you might make.

1. Change the cycle times.

2. Change the decision rule.

3. Provide a green cycle for the side street on some minimum schedule even if no cars have been detected. (There might be sports cars!)

4. Provide yellow caution lights on one or both streets.

5. Modify the program so that it doesn't lose a car that arrives during the caution cycle and then stops. (Recall that the main program loop sets CARS$WAITING to zero.)

6. Provide for an interrupt from a sensor in the main street as well as the side street. Keep moving averages on the traffic rates for the last five minutes on both streets, and adjust the cycle times to reflect relative demand. (This is a fairly sizeable project.)

APPENDIX I

INTELLEC SERIES II SYSTEM CONFIGURATIONS

MODEL 210

The Model 210 provides you with the minimum system required for the rapid and efficient development of microcomputer software, while allowing you the option of easily upgrading to a diskette-based system as your performance needs and budget allow. The Model 210's new ROM-based Editor/Assembler combination allows the development of small 8080 or 8085 programs completely in RAM memory — minimizing your usage of paper tape. An optional MCS-48™ ROM Assembler/Editor provides the same capability for the Intel MCS-48 family of single-chip microcomputers. The compact new system has 32K bytes of RAM, 24K bytes of ROM and its own microprocessor. A self-test diagnostic capability is built into the system. The Model 210 interfaces to your own terminal to get you started on your microcomputer development project — with a minimum of inconvenience and an extremely low price!

MODEL 220

The Model 220 provides you complete access to a variety of essential microcomputer development tools while using an absolute minimum of valuable laboratory bench space. Its unique packaging combines a 2000-character CRT, full-sized 256K-byte floppy diskette drive and 6-slot MULTIBUS cardcage in a single, compact unit. The design is made possible by extensive use of Intel's high-technology LSI microcomputer components, including an 8080A CPU, 8271 floppy disk controller, 8275 CRT controller and 16K RAM memories. The Model 220 has 32K bytes of RAM and 4K bytes of ROM as standard equipment. Powerful ISIS-II Diskette Operating System software with its relocating 8080/8085 assembler provides you with the perfect environment for a medium-scale system development effort. And the Model 220 interfaces directly to all the Intel In-Circuit and "In-System" Emulator modules.

MODEL 230

The most powerful member of the new Intellec Series II family is the Model 230. It includes two double density floppy diskettes and 64K bytes of RAM, an integrated CRT display and a detachable, typewriter-style keyboard with upper and lower case characters and cursor controls. The powerful ISIS-II Diskette Operating System and its relocatable and linkable software is standard with the new system, allowing you the use of Intel's high-level programming languages — including PL/M-80 and FORTRAN 80, plus the industry's most comprehensive line of macro assemblers. More than 1 million bytes of on-line diskette storage is included, and the system will support up to 2.5 million total bytes. The standard Intellec System Monitor, provided in ROM memory, contains a "Self-Test" system diagnostic. Interfaces are provided for a printer, paper tape reader/punch and universal PROM programmer. The system is compatible with all Intellec and MULTIBUS™ modules. The Model 230 provides access to all the tools needed for microprocessor-based development work . . . software development tools including editors, assemblers, compilers and debuggers . . . and system development tools including all In-Circuit Emulators plus the world's first "In-System Emulator," ICE-85™ and its 18-channel External Trace Module.

Les Watten. Smith

APPENDIX II

INTELLEC SERIES II AND RELATED DOCUMENTATION

A Guide to PL/M Programming for Microcomputer Applications, by Daniel D. McCracken, Addison-Wesley, 1978.

Intellec Series II Model 210 User's Guide, 98-558, which describes the use of the Model 210.

Intellec Series II Installation and Service, 98-557, which describes how to install all models and options of Intellec Series II, and how to perform minor service (including operation of diagnostic routines).

Intellec Series II Hardware Reference Manual, 98-556, which describes the operation of the system modules and includes high-level functional descriptions of each.

Intellec Series II Schematic Drawings, 98-554, which contains the schematic drawings for Models 210, 220, and 230.

ISIS-II User's Guide, 98-306, which describes the operation of the diskette operating system.

8080/8085 Assembly Language Programming Manual, 98-301, which describes the assembly language for the MCS-80/85 family of microprocessors.

ISIS-II 8080/8085 Assembler Operator's Manual, 98-292, which describes how to assemble an MCS-80/85 assembly language program under ISIS-II.

MCS-48/UPI-41 Assembly Language Manual, 98-255, which describes the assembly language for the MCS-48 and UPI-41 family of microprocessors and describes how to assemble programs written in those languages.

PL/M-80 Programming Manual, 98-268, which describes the source statements of the PL/M-80 language.

ISIS-II PL/M-80 Compiler Operator's Manual, 98-300, which describes how to compile a PL/M-80 program using ISIS-II.

FORTRAN-80 Programming Manual, 98-481, which describes the source statements of the 8080/8085 ANS FORTRAN compiler, which implements the FORTRAN-77 ANS standard.

ISIS-II FORTRAN-80 Compiler Operator's Manual, 98-480, which describes how to compile a FORTRAN-80 program using ISIS-II.

ICE-80 Operator's Manual, 98-185, which describes the installation and use of the ICE-80 in-circuit emulation module.

ICE-85 Operator's Manual, 98-463, which describes the installation and use of the ICE-85 in-circuit emulation module.

ICE-48 Operator's Manual, 98-464, which describes the installation and use of the ICE-48 in-circuit emulation module.

Contact Intel Literature Department, 3065 Bowers Ave., Santa Clara, CA 95051 for ordering information.